ABSTRACT

Title of dissertation: OPTIMIZING FOR A MANY-CORE ARCHITECTURE WITHOUT COMPROMISING EASE OF PROGRAMMING

George C. Caragea,
Doctor of Philosophy, 2011

Dissertation directed by: Professor Uzi Vishkin,
Department of Computer Science and
Department of Electrical and Computer Engineering

Faced with nearly stagnant clock speed advances, chip manufacturers have turned to parallelism as the source for continuing performance improvements. But even though numerous parallel architectures have already been brought to market, a universally accepted methodology for programming them for general purpose applications has yet to emerge. Existing solutions tend to be hardware-specific, rendering them difficult to use for the majority of application programmers and domain experts, and not providing scalability guarantees for future generations of the hardware.

This dissertation advances the validation of the following thesis: *it is possible to develop efficient general-purpose programs for a many-core platform using a model recognized for its simplicity.* To prove this thesis, we refer to the eXplicit Multi-Threading (XMT) architecture designed and built at the University of Maryland. XMT is an attempt at re-inventing parallel computing with a solid theoretical foundation and an aggressive scalable design. Algorithmically, XMT is inspired by the PRAM (Parallel Random Access Machine) model and the architecture design is focused on reducing inter-task communication and synchronization overheads and providing an easy-to-program parallel model.

This thesis builds upon the existing XMT infrastructure to improve support for efficient execution with a focus on ease-of-programming. Our contributions aim at reducing the programmer's effort in developing XMT applications and improving the overall performance. More concretely, we: (1) present a work-flow guiding programmers to produce efficient parallel solutions starting from a high-level problem; (2) introduce an analytical performance model for XMT programs and provide a methodology to project running time from an implementation; (3) propose and evaluate RAP – an improved resource-aware compiler loop prefetching algorithm targeted at fine-grained many-core architectures; we demonstrate performance improvements of up to 34.79% on average over the GCC loop prefetching implementation and up to 24.61% on average over a simple hardware prefetching scheme; and (4) implement a number of parallel benchmarks and evaluate the overall performance of XMT relative to existing serial and parallel solutions, showing speedups of up to 13.89x vs. a serial processor and 8.10x vs. parallel code optimized for an existing many-core (GPU). We also discuss the implementation and optimization of the Max-Flow algorithm on XMT, a problem which is among the more advanced in terms of complexity, benchmarking and research interest in the parallel algorithms community. We demonstrate better speed-ups compared to a best serial solution than previous attempts on other parallel platforms.

OPTIMIZING FOR A MANY-CORE ARCHITECTURE WITHOUT COMPROMISING EASE OF PROGRAMMING

by

George Constantin Caragea

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2011

Advisory Committee

Professor Uzi Vishkin, Chair
Professor Rajeev Barua
Professor Alan Sussman
Professor Chau-Wen Tseng
Professor William Dorland, Dean's representative

To my parents Carmen and Doru

Acknowledgments

I would like to express my gratitude to my advisor, Prof. Uzi Vishkin, for the relentless guidance that he provided, for his constant but gentle supervision and not least for his infinite patience. I have been fortunate to have the opportunity to learn from him invaluable lessons about what it means to be a successful researcher.

I would also like to thank my co-advisor, Prof. Rajeev Barua for providing me with his outstanding advice and for helping me navigate often unfamiliar territory. I am very grateful to him for always believing in me and having my best interests in mind, and making sure I did not lose perspective.

I am also grateful to my Ph.D. dissertation committee members, Prof. Alan Sussman, Prof. Chau-Wen Tseng and Prof. William Dorland for the time, effort and comments that improved this dissertation.

I would like to thank present and past members of the XMT team for tirelessly working to make sure all the pieces of the project fit together, for the stimulating discussions, enormous help writing and proof-reading papers and their warm friendship: Fuat Keceli, Alexandros Tzannes, Xingzhi Wen, Aydin Balkan, Beliz Saybasili, Michael Horak, Mary Kiemb, James Edwards, Mike Detwiler.

It is almost impossible to describe the gratitude that I feel towards my my parents Carmen and Doru Caragea and my sister Petruţa Caragea for unconditionally believing in me and supporting my decisions. I am extremely lucky to have such a wonderful family.

I am also deeply thankful to my soon-to-be wife Mariko Sweetnam for being incredibly supportive and always being by my side. Mariko helped me regain clarity when the world seemed dark and uncertain, and motivated me to keep fighting when I was ready to surrender. I would not have made it so far without her.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The number of transistors on chip has been growing rapidly in the last decades, closely folowing the exponential rate predicted by Moore's Law [Moo65]. Concurrently with the arrival of the billion-transistor chip era, the industry happened to be facing a slow down in clock rate improvement caused by power and thermal constraints. Driven by these limits, there has been a growing interest in parallel computing. In conjunction, the renewed ongoing expansion in the demands of scientific and commercial computing workloads also contributed to this shift.

Multi-core processors are on the road-map of all the hardware vendors for the foreseeable future. These systems are attractive to application developers because of their impressive peak computational potential and (in several cases) their energy-efficient processing. However, the experience has been that simply adding more cores does not necessarily improve single-task performance.

To date, the outreach of parallel computing has fallen short of historical expectations. This has primarily been attributed to programmability shortcomings of parallel computers. A broad based system redesign encompassing the areas of programming languages, compilers and hardware (among others) is needed in order to make parallel computing accessible (see e.g. [Sni08]).

The eXplicit Multi-Threading (XMT) architecture designed and built at the University of Maryland [VDBN98, NNTV01, WV08a, Wen08] is an attempt at re-inventing parallel computing with a solid theoretical foundation and an aggressive scalable design. The overall approach is focused on reducing inter-task communication and synchronization overheads and providing an easy-to-program parallel model. Algorithmically, XMT is inspired by the PRAM (Parallel Random Access Machine) model (see e.g. [KR90, EG88, JáJ92, Vis07]), which achieved wide recognition as the leading parallel model in the algorithms community. PRAM algorithms are indeed simple, prescribing: (a) a sequence of rounds, and (b) for each round, the operations that can be executed concurrently, assuming unlimited hardware and unit time for memory accesses. This simple abstraction allows the

programmer to focus only on expressing what can be executed in parallel at each point, rather than the myriad implementation concerns that are interwoven into popular parallel programming models such as OpenMP or CUDA. The vast PRAM body of research (second in volume only to serial algorithms) includes highly effective parallel algorithms even for many "non-regular" tasks, such as those including graph traversal and divide-and-conquer.

XMT is a shared memory parallel architecture with a heavy duty serial processor and many lightweight parallel processors called Thread Control Units (TCUs). An overview of XMT is provided in Chapter 2, while a detailed description of the implementation of the architecture can be found in [Wen08]. Some highlights of the architecture include:

- fast scalable hardware supported synchronization
- hardware support for thread scheduling for load balance and low overheads
- low-latency, high-bandwidth on-chip interconnection network
- PRAM-inspired Uniform Memory Access (UMA) shared memory model

Previous work on XMT has demonstrated improved performance of a 64-core FPGA-based prototype when compared to a modern serial processor on both regular and non-regular, hard to parallelize codes. [WV08a] presents speed-ups on all benchmarks used, including 8.56x on matrix multiplication, 1.62x on quick-sort, 1.88x on breadth-first search in graphs, 1.74x on longest path in directed acyclic graphs (DAGs), 2.13x on array summation, 2.06x on array compaction, 1.57x on binary tree searching and 2.74x on convolution. In [GV06], speedups of up to 100x on gate-level circuit simulations are observed when using a 1024-core simulated configuration.

These studies have brought initial proof of the viability and competitiveness of the architecture design and programming model. To obtain these speedups, expert programmers provided hand-tuned optimized code for the XMT platform, which required in-depth knowledge of both the application and the architecture details. This thesis builds upon this infrastructure to improve support for efficient execution but with a focus on ease-of-programming. More specifically, our contributions aim at reducing the programmer's effort in developing XMT applications and improving the overall performance. We introduce the concrete contributions of this thesis next.

1.1 Contributions

This dissertation advances the validation of the following thesis: *it is possible to develop efficient general-purpose programs for a many-core platform using a model recognized for its simplicity.*

We focus on two aspects of this claim: ease of programming and performance. Towards the first goal, we provide guidance for programmers interested in producing an efficient parallel solution to a given problem, as well as compiler optimizations that are transparent to the programmer but are essential for performance. In regards to the latter, we describe experiments conducted to compare our solution to existing serial and parallel architectures, and show evidence of XMT's competitiveness.

More concretely, this thesis presents several improvements within the XMT project. These improvements are crucial advances towards proving the viability of XMT as a general-purpose many-core solution. These contributions are:

- an easy to follow programmer's *workflow* from a parallel algorithmic model (PRAM) to efficient parallel executable program;
- an analytical performance model for parallel programs written for the XMT platform;
- an essential compiler optimization: resource-aware prefetching;
- architectural optimization to allocate optimum amount of resources for prefetching;
- performance comparison with a serial architecture and with existing many-core (GPU) on irregular workloads, culminating in a problem which is among the more advanced in terms of complexity, benchmarking and research interest in the parallel algorithms community, Maximum-Flow in networks.

This thesis is organized as follows: Following this introduction, we present an overview of the XMT Many-Core Platform in Chapter 2. Chapter 3 introduces a workflow guiding programmers from a problem description to an efficient parallel program, together with an analytical performance model and several cases studies. Next, in Chapter 4 we describe and evaluate the Resource-Aware Prefetching algorithm, a compiler optimization targetted at many-core architectures with limited per-core resources. Chapter 5 compares the XMT architecture with both existing serial and many-core architectures with a focus on

irregular benchmarks. In Chapter 6, we describe and discuss an XMT implementation for the Max-Flow algorithm. We conclude in Chapter 7.

Chapter 2

The XMT Many-Core Platform

Following the high level of interest related to the PRAM model in the theoretical computer science community, researchers have tried to approximate the theoretical performance of the PRAM using multi-chip parallel computing. Such attempts included the NYU-Ultracomputer [GGK+82] and the Tera/Cray MTA [ACC+90] in the 1980s and the SB-PRAM [BBF+97, KKT01, PBB+02] in the 1990s. However, the works [CKP+93, CGS97] show that the high latency, and even more importantly the limited bandwidth among the processors, make this goal difficult to accomplish using 1990s technology.

The XMT PRAM-On-Chip project at the University of Maryland is based on the observation that the fast growing number of transistors makes it possible to build a PRAM on a single chip. When multiple cores reside on the same chip, they can be connected with a very high-bandwidth, low-latency network, and the communication overhead among them can be significantly reduced compared to the multi-chip parallel computers.

In this chapter, we first review the PRAM algorithmic model, followed by an overview of the XMT architecture and programming model. This will serve as the framework for the contributions discussed in the rest of this thesis.

2.1 The PRAM Algorithmic Model

The PRAM model of computation [JáJ92, KR90, EG88, Vis07] assumes that any number of concurrent accesses to a shared memory take the same time as a single access. PRAM provides an intuitive abstraction for developing parallel algorithms; this led to an extremely rich algorithmic theory (second in magnitude only to its serial counterpart).

A PRAM consists of p synchronous processors communicating through a global shared memory accessible in unit time from each of the processors. An algorithm in the PRAM model is described as a sequence of parallel time units, or rounds; each round consists of exactly p instructions to be performed concurrently, one per each processor.

There are a variety of rules for resolving access conflicts to the same shared memory

location. The most common are exclusive-read exclusive-write (EREW), concurrent-read exclusive-write (CREW), and concurrent-read concurrent-write (CRCW), giving rise to several PRAM models. An EREW PRAM does not allow simultaneous access by more than one processor to the same memory location for read or write purposes, while a CREW PRAM allows concurrent access for reads but not for writes, and a CRCW PRAM allows concurrent access for both reads and writes. We shall assume that in a concurrent-read concurrent-write model, reads are resolved before writes and an arbitrary processor among the processors attempting to write into a common memory location, succeeds. This is called the Arbitrary CRCW rule. There are two alternative CRCW rules: (i) By the Priority CRCW rule, the smallest numbered, among the processors attempting to write into a common memory location, actually succeeds. (ii) The Common CRCW rule allows concurrent writes only when all the processors attempting to write into a common memory location are trying to write the same value.

Design of an efficient parallel algorithm for the PRAM model would seek to optimize the total number of operations the algorithms performs ("work") and its parallel time ("depth") assuming unlimited hardware. Given a PRAM algorithm described in the Work-Depth framework, Chapter 3 introduces a workflow for deriving an XMT program written in XMTC, along with ways to reason about performance guarantees similar to serial computing.

2.2 The XMT Architecture

The primary goal of the eXplicit Multi-Threading (XMT) on-chip general-purpose computer architecture (e.g. [VDBN98, NNTV01, WV08a]) is improving single-task performance through parallelism. XMT was designed from the ground up to capitalize on the huge on-chip resources becoming available with new fabrication technologies. It is meant to leverage the vast body of knowledge of PRAM algorithmics, and the latent, though not widespread, familiarity with it.

The XMT architecture, depicted in Fig. 2.1, includes an array of lightweight cores called Thread Control Units (TCUs) and a serial core with its own cache (Master TCU). The processor includes several clusters of TCUs connected by a high-throughput mesh-of-trees (MOT) interconnection network [BHQV07]; an instruction and data broadcast mechanism;

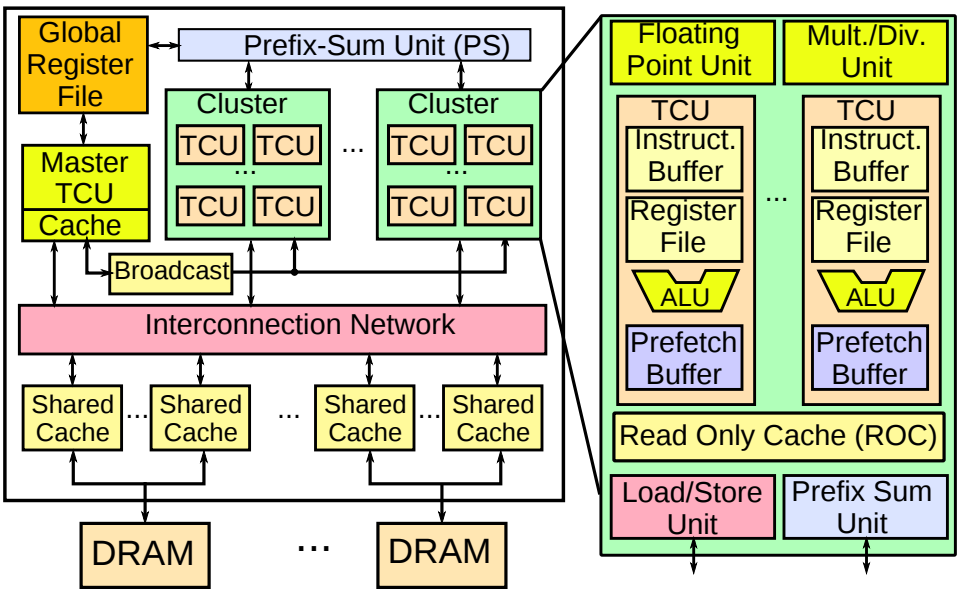


Figure 2.1: XMT architecture overview

a global register file (GRF); a prefix-sum unit (PS). The first level of cache is shared and partitioned into mutually-exclusive cache modules sharing several off-chip DDR2 DRAM memory channels. The TCU Load-Store unit applies a hashing function on each address to avoid memory hotspots. Cache modules handle concurrent requests and provide buffering and request reordering to achieve better DRAM bandwidth utilization. Within a cluster, a compiler-managed Read-Only Cache (ROC) is used to store constant values across all threads. TCUs include lightweight ALUs, but the more expensive Multiply/Divide (MDU) and Floating Point Units (FPU) units are shared by all TCUs in a cluster.

An extensive description of the XMT architecture is presented in [Wen08]. A first commitment to silicon, a 64-core FPGA prototype, was reported and evaluated in [Wen08, WV07, WV08a].

2.3 The XMT Memory Hierarchy

Figure 2.2 gives an overview of the XMT memory hierarchy while operating in parallel mode. The compiler can issue the following types of instructions to control the movement of data between the registers, TCU prefetch buffers, Read-Only Buffers, L1 shared cache and the off-chip DRAM:

- **Regular load.** A regular read instruction will first check the TCU prefetch buffers,

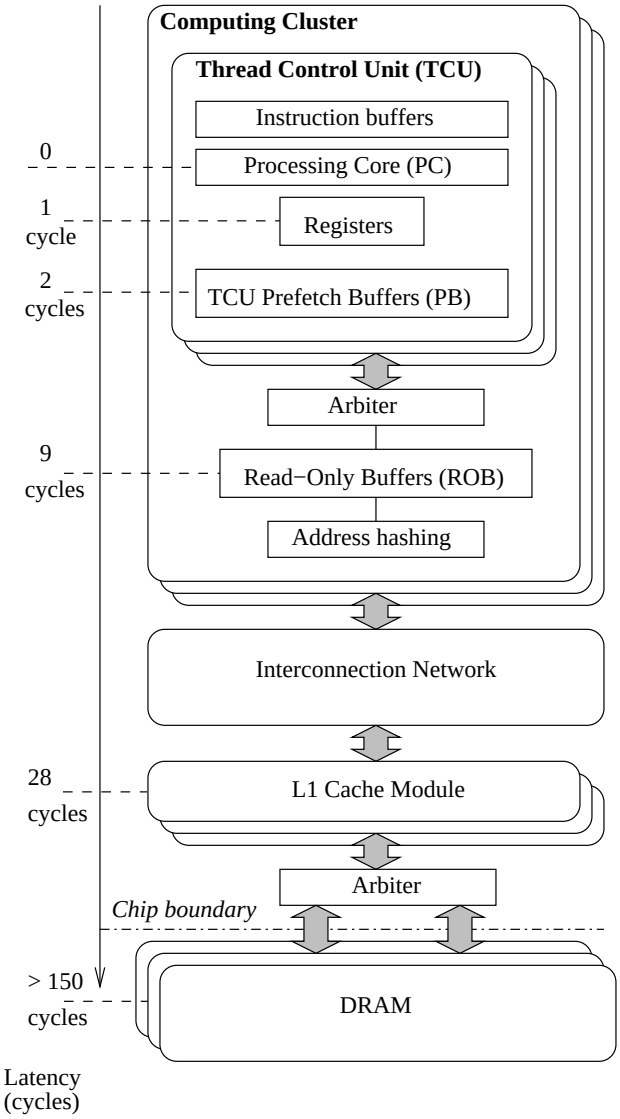


Figure 2.2: The XMT Memory Hierarchy while in parallel mode. On the right side the latency from the processing core (PC) to each level of the memory hierarchy is shown. Some elements, such as the Master TCU, which operates in serial mode, the global register file and the prefix-sum unit, are omitted for simplicity.

then the Read-Only buffers, then travel through the interconnection network to the shared cache, and finally to DRAM. If the requested memory location is found in any of these modules, the request is served from that module, and the reply is sent straight back to the issuing TCU.

- **Cacheable load.** It operates in the same way as a regular load, to locate the contents of a memory location. The only difference is that the result is also copied to the corresponding cluster's Read-Only buffer before being sent back to the issuing TCU.

- **TCU Prefetch Instructions.** A prefetch-to-TCU instruction reserves one location (word) in the TCU prefetch buffer array and sends the request to the lower levels. The TCU proceeds executing the next instruction without waiting for the prefetch request to complete. One such prefetch instruction brings one word to the TCU prefetch buffers, following the same order as a regular load for locating the memory content. A subsequent read request first checks the TCU prefetch buffers, and if a valid or pending corresponding entry is found, the reply is sent back to the processing core, possibly after waiting for the prefetch request to return from the lower levels.

- **Cacheable TCU Prefetch Instruction.** Operates just like a TCU prefetch instruction, but it also writes the value of the fetched memory location to the corresponding cluster's Read-Only Buffer.

- **Regular (Blocking) Stores.** A store instruction first invalidates any matching TCU prefetch buffer locations at the issuing TCU, then it is sent straight through the interconnection network to the shared cache. Note that stores do not invalidate Read-Only buffer locations, and therefore it is the responsibility of the compiler to ensure correctness when populating the Read-Only buffers. The issuing TCU waits for an acknowledgment from the shared cache before proceeding to execute the next instruction.

- **Non-blocking Stores.** A non-blocking store operates the same way as a regular store, but the TCU proceeds with executing the next instruction in the next clock cycle after issuing the store, without waiting for the acknowledgment from the shared cache.

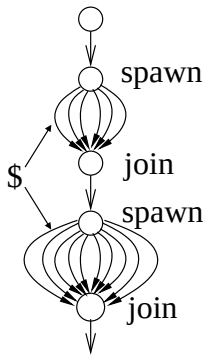


Figure 2.3: The XMT execution model: switching between serial and parallel modes.

- **Store flush.** Each TCU keeps a counter of all the outstanding non-blocking stores it has issued. A store-flush instruction is available in the ISA, which allows the TCU to block until all the store instructions have been acknowledged by the shared cache. This instruction can be used to implement the XMT Memory Consistency Model and provide guarantees to the programmer, as discussed in Section 3.1.6.

2.4 XMT Programming Model and XMTC

On XMT, as well as in other parallel programming frameworks (e.g. OpenMP, Intel TBB, NVIDIA CUDA), the programmer is encouraged to express all the parallelism available in the application; a scheduler is used to allocate the parallel units of work (threads) to the physical execution units. Execution consists of a succesion of serial sections, where only the MTCU is active, and parallel sections where all the TCUs execute in parallel.

To support this paradigm, XMT uses a new programming language, XMTC, which was designed to provide an easy mapping for the programmer from PRAM algorithmics structures, as well as straightforwad translation to low-level operations supported by the XMT hardware. XMTC [BV06] is an extension of the C programming language. The new primitives in the XMTC language and the XMT programming model are:

Spawn Instruction. Starts a parallel section. Accepts as parameter the number of parallel threads to start. The directive will spawn multiple virtual threads and execute the ensuing code in parallel. The threads are usually short and the execution switches frequently between serial and parallel modes as pictured in Figure 2.3. The `spawn()` instruction provides support for a PRAM-like *pardo* loop.

```
int A[N],B[N],C[N];
int base=0;
spawn(0,N-1) {
    int inc=1;
    if (B[$]!=0) {
        ps(inc,base);
        C[inc]=A[$];
    }
}
```

Figure 2.4: XMTC program example: Array Compaction

Thread-id. A reserved identifier inside a parallel section accessed using the $ symbol. Evaluates to the unique thread index. This allows SPMD style programming.

Prefix-sum Instruction. The prefix-sum instruction defines an atomic operation. First assume a global variable B, called base, and a local variable R, called increment, the result of a prefix-sum is: (i) B gets the value $B + R$, and (ii) R gets the original value of B. While, the basic definition of prefix-sum follows the fetch-and-add of the NYU-Ultracomputer [GGK+82], XMT uses a fast parallel hardware implementation if R is from a small range (e.g., one bit) and B can fit one of a small number of global registers[Vis97]; otherwise, prefix-sums are obtained using a prefix-sum-to-memory instruction; in the latter case, prefix-sum implementation involves queuing in memory.

XMTC implements an a hybrid of so-called arbitrary CRCW (Concurrent Read Concurrent Write) and QRQW (Queue Read Queue Write) [GMR98] memory access models. The Arbitrary CRCW PRAM dictates that if multiple threads attempt to update the same memory location simultaneously, then an arbitrary one will succeed. An algorithm designed with this property in mind permits each thread to progress at its own speed from its initiating spawn to its terminating join, without ever having to wait for other threads; that is, no thread busy-waits for another thread. We call this "independence of order semantics" (IOS). The prefix-sum instruction provides low overhead thread coordination, and is useful for implementing concurrent writes, in accordance to the IOS paradigm.

Figure 2.4 presents an example of XMTC code.The elements of array A which correspond to non-zero elements of B are copied into an array C. The order is not necessarily preserved. After the execution of `ps(inc,base)` statement, the `base` variable is in-

creased by `inc` and the `inc` variable gets the original value of `base`, as an **atomic** operation.

Chapter 3

Programmer's Workflow for Advancing From PRAM to XMT

The current chapter introduces a workflow for translating a PRAM algorithm into an efficient parallel program for explicit multi-threading (XMT). Given a text on PRAM algorithms as well as our XMT system tool-chain, comprising a compiler and an instance of the hardware, the methodology links the algorithms and the system.

More concretely, we revisit a widely used methodology for advancing parallel algorithmic thinking into parallel algorithms and extend it into a methodology for advancing parallel algorithms to XMT (or "PRAM-On-Chip") programs. This chapter also presents a performance cost model for XMT. It uses as complexity metrics the **length of sequence of round-trips to memory** (LSRTM) and **queuing delay** (QD) from memory access queues, in addition to standard PRAM computation costs of work and depth. One of the contributions of this chapter is highlighting the importance of LSRTM in determining performance.

It was unavoidable to have XMT architecture choices impact the performance cost model being proposed. However, the proposed model is quite general, as it abstracts away most low-level details. The model and methodology are quite robust, and can be of interest beyond the context of particular XMT architecture choices.

The main contributions of this chapter are as follows.

- We present a programmer's workflow for converting PRAM algorithms to PRAM-on-chip programs.
- We introduce a performance model used in developing a PRAM-On-Chip program, with a particular emphasis on a certain complexity metric, the length of the sequence of round trips to memory (LSRTM). While the PRAM algorithmic theory is pretty advanced, many more practical programming examples need to be developed. For standard serial computing, examples for bridging the gap between algorithm theory and practice of serial programming abound, simply because it has been practiced for so long. See also [Ski97]. A similar knowledge base needs to be developed for parallel computing.

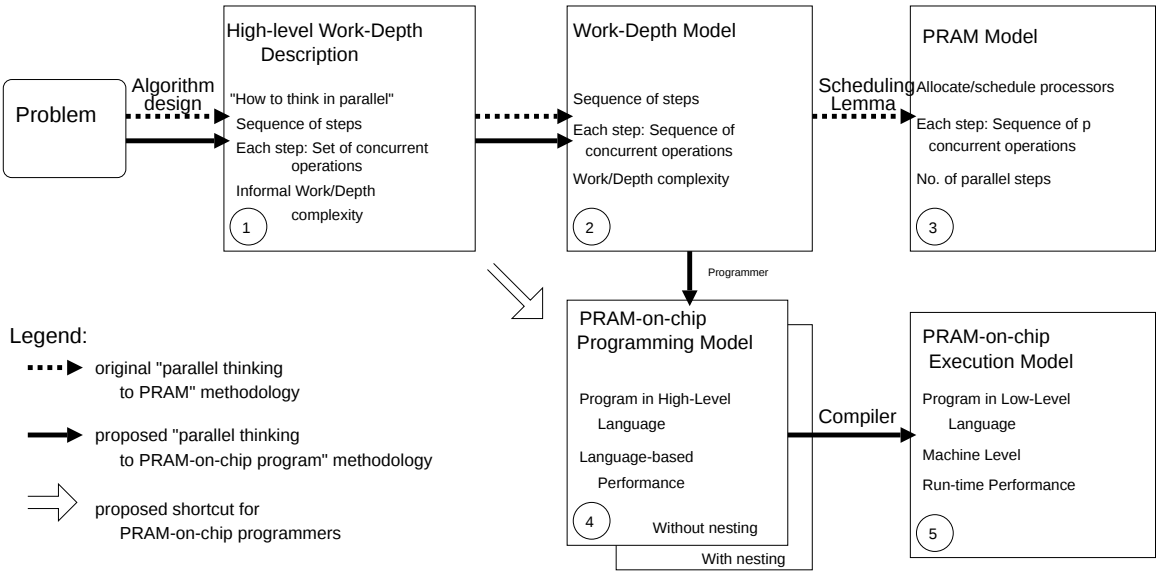


Figure 3.1: Models for advancing PRAM algorithms to parallel programs

- We provide a few initial examples of applying the methodology to different problems.
- We discuss alternatives to the strict PRAM model that by further suppressing of details provide (even) easier-to-think frameworks for parallel algorithms and programming development.

3.1 Models in the Workflow

Given a problem, this chapter proposes a "recipe" for developing an efficient XMT program from concept to implementation. In particular, we present the stages through which such development needs to pass.

Figure 3.1 depicts the proposed methodology. For context, the figure also depicts the widely used Work-Depth methodology for advancing from concept to a PRAM algorithm, namely, the sequence of models $1 \rightarrow 2 \rightarrow 3$ in the figure. There are two drawbacks to this methodology. First, it targets a theoretical model (PRAM), and not a real execution platform, limiting the usefulness to asymptotical complexity analysis. And second, it is quite difficult for a programmer to describe an algorithm in the PRAM model, which requires specifying the task of each processor at each time step; it is significantly more effective to have the programmer target a higher level abstraction, and let the compiler, run-time

system and/or hardware handle the additional scheduling complexities.

For developing a XMT implementation, we propose following the sequence of models $1 \rightarrow 2 \rightarrow 4 \rightarrow 5$, as follows.

- Given a specific problem, an algorithm design stage will produce a High-Level description of the parallel algorithm, in the form of a sequence of steps, each comprising a *set* of concurrent operations (box 1). In a first draft, the set of concurrent operations can be implicitly defined.

- This first draft is refined to a sequence of steps each comprising now an *ordered sequence* of concurrent operations (box 2). Such formal Work-Depth description fully spells out how to advance in a given step, whose sequence of concurrent operations include j operations indexed by integers from 1 to j, from each index i where $1 \leq i \leq j$, to an operation.

- Next, the programming effort amounts to translating this description into a single-program multiple-data (SPMD) program using a high-level XMT programming language (box 4).

- From this SPMD program, a compiler will transform and reorganize the code to achieve the best performance in the target XMT execution model (box 5).

As an XMT programmer gains experience, he/she will be able to skip box 2 (the Work-Depth model) and directly advance from box 1 (high-Level Work-Depth description) to box 4 (high-level XMT program). We also demonstrate some instances where it may be advantageous to skip box 2 because of some features of the programming model (such as some ability to handle nesting of parallelism). In Figure 3.1 this shortcut is depicted by the arrow $1 \rightarrow 4$. We start by elaborating on each model, and follow by demonstrating the methodology on a few problems.

3.1.1 PRAM Model

The PRAM algorithmic model is reviewed in Section 2.1. PRAM algorithms are essentially prescribed as (a) a sequence of rounds, and (b) for each round, up to p processors can execute concurrently. The performance objective is minimizing the number of rounds. The PRAM parallel algorithmic approach is well-known and has never been seriously

challenged by any other parallel algorithmic approach on ease of thinking, or wealth of knowledge-base. However, PRAM is a strict formal model. A PRAM algorithm must prescribe for each and every one of its p processors the instruction that the processor executes at each time unit in a detailed computer-program-like fashion, which can be quite demanding. The PRAM algorithms theory mitigates this using the work-depth (WD) methodology, discussed next.

3.1.2 The Work-Depth Methodology

The Work-Depth methodology for designing PRAM algorithms, introduced in [SV82], has been quite useful for describing parallel algorithms and reasoning about their performance. For example, it was used as the description framework in [JáJ92]. The methodology guides algorithm designers to optimize two quantities in a parallel algorithm: *depth* and *work*. Depth represents the number of steps the algorithm would take if unlimited parallel hardware was available, while work is the total number of operations performed, over all parallel steps.

The methodology comprises of two steps: (i) first, produce an informal description of the algorithm in a high-level work-depth model (HLWD), and (ii) refine this description into a fuller presentation in a model of computation called Work-Depth. These two models are described next.

3.1.3 High-Level Work-Depth Description

A HLWD description consists of a succession of parallel rounds, each round being a *set* of any number of instructions to be performed concurrently. Descriptions can come in several flavors, and even implicit descriptions, where the number of instructions is not obvious, are acceptable.

Example: Input: An undirected graph $G(V, E)$ and a source node $s \in V$; the length of every edge in E is 1. Find the length of the shortest paths from s to every node in V. An informal work-depth description of a parallel *breadth-first search (BFS)* algorithm can look as follows. Suppose that the set of vertices V is partitioned into layers. Layer L_i includes all vertices of V whose shortest path from s have i edges. The algorithm works in iterations. In iteration i, layer L_i is found. Iteration 0: node s forms layer L_0. Iteration i, $i > 0$: Assume inductively that layer L_{i-1} was found. In parallel, consider all the edges (u, v) that have

an endpoint u in layer L_{i-1}; if v is not in a layer L_j, $j < i$, it must be in layer L_i. As more than one edge may lead from a vertex in layer L_{i-1} to v, vertex v is marked as belonging to layer L_i based on one of these edges using the arbitrary concurrent write convention. This ends an informal, high-level work-depth verbal description.

A pseudo-code description of an iteration of this algorithm could look as follows:

```
for all vertices v in L(i) pardo
  for all edges e=(v,w) pardo
    if w unvisited
       mark w as part of L(i+1)
```

The above HLWD description challenges us to find an efficient PRAM implementation for an iteration. In particular, given a p-processor PRAM, how to allocate processors to tasks to finish all operations of an iterations as quickly as possible. A more detailed description in the Work-Depth model would need to address these issues.

3.1.4 Work-Depth Model

In the Work-Depth model an algorithm is described in terms of successive time steps, where the concurrent operations in a time step form a sequence; each element in the sequence is indexed from 1 to the number of operations in the step. The Work-Depth model is formally equivalent to the PRAM. For example, a work-depth algorithm with $T(n)$ depth (or time) and $W(n)$ work runs on a p processor PRAM in at most $T(n) + \lfloor \frac{W(n)}{p} \rfloor$ time steps. The simple equivalence proof follows Brent's scheduling principle, which was introduced in [Bre74] for a model of parallel model of computation that was much more abstract than the PRAM (counting arithmetic operations, but suppressing anything else).

Example (continued): We only note here the challenge for coming up with a Work-Depth description for the BFS algorithm: to find a way for listing in a single sequence all the edges whose endpoint is a vertex at layer L_i. In other words, the Work-Depth model does not allow us to leave nesting of parallelism, such as in the pseudo-code description of BFS above, unresolved. On the other hand XMT programming should allow nesting of parallel structures, since such nesting provides an easy way for parallel programming. It is also important to note that the XMT architecture includes some limited support for nesting of parallelism: a nested spawn can only spawn k extra threads, where k is a small integer

(e.g., $k = 1, 2, 4$ or 8); nested spawn commands are henceforth called either k-spawn, or sspawn (for single spawn). The way in which we suggest to resolve this problem is as follows. The *ideal* long term solution is: (a) allow the programmer free unlimited use of nesting, (b) have it implemented as efficiently as possible by compiler, and (c) make the programmer (especially the "performance programmer") aware of the added cost of using nesting. At the time of this writing, work is under way to add efficient support for arbitrary nesting of parallelism to XMT [TCBV10]. However, since this work is still in progress, my *tentative* short term solution is presented with an example in Section 3.3.3, which shows how to build on the support for nesting provided by the architecture. There is merit to this "manual solution" beyond its tentative role until the compiler matures. Such solution should still need to be understood (even after the ideal compiler solution is in place) by performance programmers, so that the impact of nesting on performance is clear to them.

This example actually demonstrates that the methodology can sometimes benefit from proceeding directly to the PRAM-like programming methodology, rather than make a "stop" at the Work-Depth model.

3.1.5 XMT Programming Model

A framework for a high-level programming language, the XMT programming model seeks to mitigate two goals: (i) *Programmability:* given an algorithm in the HLWD or Work-Depth models, the programmer's effort in producing a program should be minimized; and (ii) *Implementability:* effective compiler translation of the program into the XMT execution model should be feasible.

The XMT programming model is based on the XMT programming language discussed in Section 2.4. It encompasses the semantics of the language primitives used to manage parallelism (e.g. spawn, sspawn), the atomicity guarantees provided by the prefix-sum instructions (ps and psm) as well as the restrictions associated with using these instructions and the XMT memory consistency model.

The main role of the programming model is to provide support for implementing an algorithm described in the Work-Depth model using the XMTC programming language. Several aspects of this translation are discussed below.

Using Prefix-Sum Instructions. Suppose that threads 2, 3 and 5, respectively, execute concurrently the commands $ps(B, R_2)$, $ps(B, R_3)$ and $ps(B, R_5)$, respectively all relating to the same base B and the original values are $B = 0, R_2 = R_3 = R_5 = 1$. Independence of Order Semantics (IOS) allows any order of execution among the 3 prefix-sums commands, namely – any of the 6 possible permutations. The result of all 6 permutations is $B = 3$. If thread 5 precedes thread 2 that precedes thread 2, one will get $R_5 = 0$, $R_2 = 1$ and $R_3 = 2$, and if the thread order is 2, 3 and 5 then $R_2 = 0, R_3 = 1, R_5 = 2$.

We discuss two instances of using Prefix-Sum instructions to implement algorithms in the HLWD description.

- Consider the problem of *array compaction*, where a subset of marked elements of an array are to be identified to a second array, in any order. The XMTC code for this shown in Figure 2.4. The atomic prefix-sum instruction is used to compute a unique location in the destination array for every element that needs to be copied.

- In order to implement the PRAM arbitrary concurrent write convention, the programmer is guided to do the following: Each location that might be written by several threads has an auxiliary "gatekeeper" location associated with it, initialized with a known value (say 0). When a thread wants to write to the shared location, it first executes a Prefix-sum instruction (e.g., with an increment of 1) on the gatekeeper location. Only one thread gets 0 as its result; this thread is allowed to write to the shared location, while the other threads advance to their next instruction without writing. An example of this technique is discussed for the implementation of Breadth-First Search in Section 3.3.3.

Nested parallelism. A parallel thread can be programmed to initiate more threads. However, as noted in Section 3.1.4, this comes with some (tentative) restrictions and cost caveats, due to compiler and hardware support issues. As illustrated with the Breadth-First search example discussed in Section 3.3.3, nesting of parallelism could improve the programmer's ability to describe algorithms in a clear and concise way. At the time of this writing, work is under way within the XMT group for efficient, unrestricted support of nesting. See e.g. [TCBV10].

The XMT Memory Consistency Model. The memory consistency model for a parallel computing environment is a contract between the programmer and the architecture, specifying how memory actions (reads and writes) in a program appear to execute to the programmer, and specifically which value each read of a memory location may return [AG96, MPA05]. Providing a sequential consistency model (in which the result of the program is always the same as that of some sequential interleaving of the instructions in parallel threads) is over-restrictive and leads to inefficient use of resources [GGH91]. Most modern systems use more relaxed memory consistency models, which allow for more efficient code and simpler architecture design, but require the programmer's knowledge of the model.

On XMT, the execution imposes serial order on any access of two concurrent threads relative to a prefix-sum instruction with the same base. This includes prefix-sum to register and prefix-sum to memory operations. All instructions prior to such a prefix-sum instruction in the first thread to reach the prefix-sum execute before any instructions after the prefix-sum instruction in the second thread.

3.1.6 XMT Execution Model

While a XMT programmer usually has the programming model as a target, a compiler will optimize for the execution model. The execution model depends on XMT architecture choices. However, this dependence is rather minimal and should not compromise the generality of the model for other future chip-multiprocessing architectures whose general features are similar.

All the features of the XMT architecture (described in Section 2.2), as well as the XMT memory hierarchy (Section 2.3) are part of the XMT execution model. Additionally, the model can include exact or relative costs of operations, to guide the compiler (or performance programmer) to generate the most efficient code.

For illustration purposes, Table 3.1 includes a list of such operations and costs. We will use these costs as part of the performance model in Section 3.2 as well as for the examples in Section 3.3.

In the Execution model, a program could include:

- Prefetch instructions to bring data from the lower memory hierarch levels either into the shared caches or into the prefetch buffers located at the TCUs.

Operation	Cost
Round-trip to Memory – RTM ($\mathcal{R}$)	24 clock cycles
spawn	2 RTM
join	1 RTM
sspawn	1 RTM
kspawn	1 RTM
prefetch	0 RTM
memory load (not prefetched)	1 RTM
memory load (prefetched)	0 RTM
memory write (blocking)	1 RTM
memory write (non-blocking)	1 RTM
memory flush	1 RTM
ps	1 RTM, 0 queuing
psm	1 RTM, NTCU max. queuing

Table 3.1: Costs of operations in XMT Execution Model

- Broadcast instructions, where some values needed by all, or nearly all TCUs, are broadcast to all.
- Thread clustering: combining shorter virtual threads into a longer thread, discussed below in Section 3.1.6.1.
- If the programming model allows nested parallelism, the compiler will use the mechanisms supported by the architecture to implement or emulate it.

3.1.6.1 Clustering

The XMT Programming Model allows spawning an arbitrary number of virtual threads, but the architecture has only a limited number of TCUs to run these threads. In the progression from the Programming Model to the Execution Model, the compiler or runtime system often needs to make a choice between two options: (i) spawn fewer threads each effectively executing several shorter threads, and (ii) run the shorter threads as is. Combining short threads into a longer thread is called clustering and offers several advantages:

- *reduce RTMs:* one can pipeline memory accesses that had previously been in separate threads (e.g. by using loop prefecthing, as discussed in Chapter 4); this can reduce extra costs from serialization of RTMs and QDs that are not on the critical path;
- *reduce thread initiation overhead:* spawning fewer threads means reducing thread initiation overheads, i.e. the time required to start a new thread on a recently freed

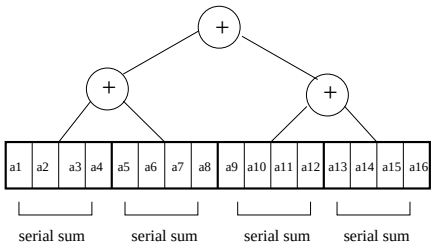


Figure 3.2: The summation algorithm with thread clustering

TCU;

- *improve pipeline usage:* similar to loop unrolling, clustering can provide opportunities for the compiler instruction scheduler to reorder instructions and reduce pipeline stalls.

Note that if the code provides fewer threads than the hardware can support, there are few advantages if any to using fewer longer threads. Also, running fewer, longer threads can adversely affect the automatic load balancing mechanism. Thus, as discussed below, the granularity of the clustering is an issue that needs to be addressed.

In some cases, clustering can be used to group the work of several threads and execute this work using a serial algorithm. For example, in the Summation algorithm the elements of the input array are placed in the leaves of a k-ary tree, and the algorithm climbs the tree computing for each node the sum of its children. However, one can instead start with an embarrassingly parallel algorithm in which they spawn p threads that each serially sum $\frac{N}{p}$ elements and then sum the p sums using the parallel summation algorithm. See Figure 3.2.

With such switch to a serial algorithm, clustering is nothing more than a special case of the accelerating cascades technique [CV86, Vis07]. For applying accelerating cascades, two algorithms that solve the same problem are used. One of the algorithms is slower than the other, but requires less work. If the slower algorithm progresses in iterations where each iteration reduces the size of the problem considered, the two algorithms can be assembled into a single algorithm for the original problem as follows: 1. start with the slower algorithm and 2. switch to the faster one once the input size is below some threshold. This often leads to faster execution than by each of the algorithms separately.

Finding the optimal crossover point between the slow (e.g., serial) and faster algorithms is needed. Also, accelerating cascades can be generalized to situations where more than two algorithms exist for the problem at hand.

We implemented initial support for clustering in the XMTC compiler for cases where the number of threads is known at runtime (i.e., where there are no nested spawns). Clustering for cases when nested spawns are used is currently under development part of the work on nesting within the XMT project. Some initial results are presented in [TCBV10]. We elaborate on both cases below.

Clustering without Nested Spawns. Suppose one wants to spawn N threads, where $N \gg p$. Instead of spawning each as a separate thread, one could trivially spawn only c threads, where c is a function of the number of TCUs, and have each one complete $\frac{N}{c}$ threads in a serial manner. Sometimes an alternative serial algorithm can replace running the N/c threads. Applying this mechanism can create a situation where most TCUs have already finished, but a few long threads are still running. To avoid this, shorter threads can be ran as execution progresses toward completion of the parallel section [NNTV01].

Clustering for single-spawn and k-spawn. In the hardware, the number of current virtual threads (either running or waiting) is broadcast to TCUs as it is updated. Assuming some system threshold, each running thread can determine whether the number of (virtual) threads scheduled to run is within a certain range. When a single-spawn is encountered, if below the threshold, the single-spawn is executed; otherwise, the thread enters a temporary spawning suspension mode and continues execution of the original thread; the thread will complete its own work and can also serially do the work of the threads whose spawning it has suspended. However, the suspension decision can be revoked once the number of threads falls below a threshold. If that occurs, then a new thread is single-spawned. Often, half the remaining work is delegated to the new thread. Clustering with k-spawn is similar. If several threads complete at the same time it will take some time to reach p running threads again, causing a gap in parallel hardware usage. This can be avoided by having a larger threshold, which would keep a set of threads ready to be started as soon as hardware becomes available.

This concludes the discussion of the models in the workflow. We proceed to discuss a performance model based on this workflow next.

3.2 Performance Modeling

The performance modeling of a program first extends the known PRAM notions of *work* and *depth*. Later, we provide a formula for estimating execution time based on these extensions.

The depth of an application in the XMT Execution model must include the following three quantities:

1. *Computation Depth*, given by the number of operations that have to be performed sequentially, either by a thread or while in serial mode.

2. *Length of Sequence of Round-Trips to Memory (or LSRTM)* which represents the number of cycles on the critical path spent by execution units waiting for data from memory. For example, a read request or prefix-sum instruction from a TCU usually causes a round-trip to memory (or RTM); memory writes in general proceed without acknowledgment, thus not being counted as round-trips, but ending a parallel section implies one RTM used to flush all the data still in the interconnection network to the memory.

3. *Queuing delay (or QD)* which is caused by concurrent requests to the same memory location; the response time is proportional to the size of the queue.

As discussed in Chapter 2, there are two types of prefix-sum instructions: prefix-sum to register `ps` and prefix-sum to memory `psm`.

The prefix-sum to register instruction (`ps`) is supported by a special hardware unit that combines calls from multiple threads into a single multi-operand prefix-sum operation. Therefore, a `ps` statement to the same base over several concurrent threads causes one roundtrip through the interconnection network to the global register file (1 RTM) and 0 queuing delay, in each of the threads.

The prefix-sum to memory (`psm`) statement is similar to `ps` except the base variable is a memory location instead of a global register. As updates to the memory location are queued, the `psm` statement costs 1 RTM and additionally has a queuing delay (QD) reflecting the plurality of concurrent threads executing `psm` with respect to the same memory location.

We can now define the XMT "execution depth" and "execution time". XMT Execution Depth represents the time spent on the "critical path" (that is, the time assuming unlimited amount of hardware) and is the sum of the computation depth, LSRTM, and QD on the critical path. Assuming that a round-trip to memory takes $\mathcal{R}$ cycles:

$$Execution\ Depth = Computation\ Depth + LSRTM \times \mathcal{R} + QD \tag{3.1}$$

Sometimes, a step in the application contains more *Work* (the total number of instructions executed) to be executed in parallel than what the hardware can handle concurrently. For the additional time spent executing operations outside the critical path (i.e., beyond the Execution Depth), the work of each parallel section needs to be considered separately. Suppose that one such parallel section could employ in parallel up to p_i TCUs. Let $Work_i = p_i * ComputationDepth_i$ be the total computation work of parallel section i. If our architecture has p TCUs and $p_i < p$, one will be able to use only p_i of them, while if $p_i \geq p$, only p TCUs can be used to start the threads, and the remaining $p_i - p$ threads will be allocated to TCUs as they become available; each concurrent allocation of p threads to p TCUs is charged as one RTM to the Execution Time, as per equation 3.2. The total time spent executing instructions outside the critical path over all parallel sections is given in equation 3.3.

$$Thread\ Start\ Overhead_i = \left\lceil \frac{p_i - p}{p} \right\rceil \times \mathcal{R} \tag{3.2}$$

$$Time\ Additional\ Work = \sum_{spawn\ block\ i} \left(\frac{Work_i}{min(p, p_i)} + ThreadStartOverhead_i \right) \tag{3.3}$$

In the last equation we do not subtract the quantity that is already counted as part of the Execution Depth. The reason is that the objective of the current work is limited to extending the work-depth upper bound $T(n) + \lfloor \frac{W(n)}{p} \rfloor$, and such double count is possible in that original upper bound as well. Adding up, the execution time of the entire program is:

$$Execution\ Time = Execution\ Depth + Time\ Additional\ Work \tag{3.4}$$

3.2.1 Clarifications of the Modeling

This chapter provided a performance modeling framework that allows weighing alternative implementations of the same algorithm where asymptotic analysis alone is insufficient. Namely, a more refined measure than the asymptotic number of parallel memory accesses was needed.

Next, we point out a somewhat subtle point. Following the path from the HLWD model to the XMT models in Figure 3.1 may be important for optimizing performance, and not only for the purpose of developing a XMT program. Bandwidth is not accounted for in the XMT performance modeling, since the on-chip XMT architecture should be able to provide sufficient bandwidth for an efficient algorithm in the Work-Depth model. The only way in which our modeling accounts for bandwidth is indirect: by first screening an algorithm through the Work-Depth performance modeling, where the model accounts for work. Now, consider what could have happened had XMT performance modeling not been coupled with Work-Time performance modeling. The program could include excessive speculative prefetching to supposedly improve performance (reduce LSRTM). The subtle point is that the extra prefetches add to the overall work count. Accounting for them in the Work-Depth model prevents this "loophole".

It is also important to recognize that the model abstracts away some significant details. The XMT hardware has a limited number of memory modules. If multiple requests attempt to access the same module, queuing will occur. While accounting for queuing to the same memory location, the model does not account for queuing accesses to different locations in the same module. Note that hashing memory addresses among modules lessens problems that would occur for accesses with high spatial locality and generally mitigates this type of "hot spots". If functional units within a cluster are shared between the TCUs, threads can be delayed while waiting for functional units to become available. The model does also not account for these delays.

Our modeling is a first-order approximation of run time. Such analytic results are not a substitute for experimental results, since the latter will not be subject to the approximations described above. In fact, we discuss some experimental results as well as a comparison between modeling and simulations in Section 3.4.

Similar to some serial performance modeling, the above modeling assumes that data

is found in the (shared) caches. This allows proper comparison to serial computing where data is found in the cache, as the number of clocks to reach the cache for XMT is assumed to be significantly higher than in serial computing; for example, our prototype XMT architecture suggests values that range between 6 and 24 cycles for a round-trip to the first level of cache, depending on the characteristics of the interconnection network and its load level; we took the conservative approach to use the value $\mathcal{R} = 24$ cycles for one RTM for the rest of this chapter. We expect that the number of clocks to access off-chip main memory should be similar to serial computing and that both for serial computing and for XMT large caches will be built.

As pointed out earlier, some of the computation work is counted twice in our Execution Time, once as part of the critical path under Execution Depth and once in the Additional Work factor. Future work could refine the analysis into a more accurate model, but with much more involved formulas. In this first version of the model, we made the choice to stop at this level of detail allowing a concise presentation while still providing relevant results.

3.3 Examples of Using the Methodology

This section presents several examples of using the programmer's framework, as well as the performance model. For each example, several algorithms are presented and compared using the analytical model.

3.3.1 Parallel Summation

Consider the problem of computing in parallel the sum of N values stored in array A. A High-Level Work-Depth description of the algorithm is as follows: in parallel add groups of k values; apply the algorithm recursively on the $\lceil N/k \rceil$ partial sums until the total sum is computed. This is equivalent to climbing (from leaves towards root) a balanced k-ary tree. An iterative description of this algorithm that fits the Work-Depth model can be easily derived from this. The parameter k is a function of the architecture parameters and the problem size N and is chosen to minimize the estimated running time.

An pseudo-code description of this algorithm in the XMT Programming Model is presented in Figure 3.3, and the full XMTC implementation is included in Appendix A.1.

```
/* Input: N numbers in the leaves of a k-ary tree in a 1D
 * array representation
 * Output: The sum of the numbers in sum[0]     */
level = 0;
while(level < log_k(N) ) {
    /* process levels of tree from leaves to root */
    level++; /* also compute current_level_start and end_index */
    spawn(current_level_start_index, current_level_end_index) {
        int count, local_sum=0;
        for(count = 0; count < k; count++)
            temp_sum += sum[k * $ + count + 1];
        sum[$] = local_sum;
    }
}
```

Figure 3.3: XMTC pseudo-code for k-ary tree summation

Note that a uni-dimensional array is used to store the complete k-ary tree, where the root is stored at element 0, followed by the k elements of the second level from left to right, then the k^2 elements of the second level and so on.

We determined the following factors about the execution by examining the code in Appendix A.1. For each iteration of the algorithm, the k partial sums from a node's children have to be read and summed. Prefetching can be used to pipeline the memory accesses for this operation, thus requiring only one round-trip to memory (RTM). An additional RTM is needed to flush all the data to memory at the end of each step. There are no concurrent accesses to the same memory location in this algorithm, thus the queuing delay (QD) is zero. By accounting for the constant factors in my XMTC implementation from Appendix A.1, we determined the Computation Depth to be $(3k + 9)\log_k N + 2k + 33$, given that $\log_k N$ iteration are needed.

To compute the additional time spent executing outside the critical path (in saturated regime), we determined the computation per tree node to be $C = 3k+2$ and the total number of nodes processed under this regime to be $Nodes_{sat}$ as in Figure 3.4. An additional step is required to copy the data into the tree's leaves at the start of the execution.

This determines the Execution Work, Additional Work an the Thread Start Overhead terms of the XMT execution time:

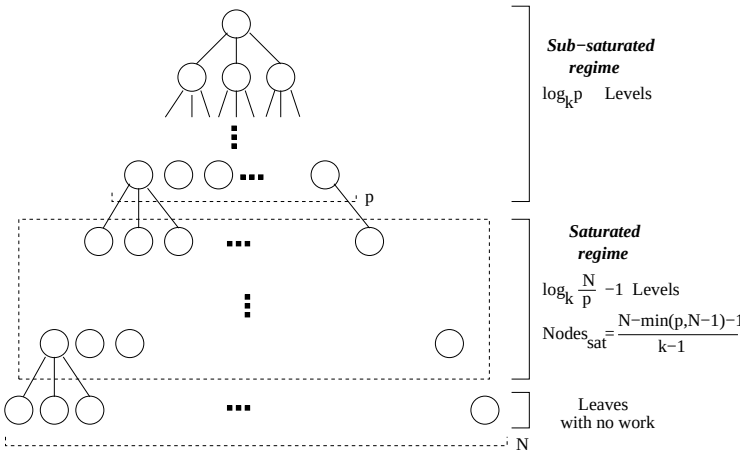


Figure 3.4: The $\log_k p$ levels of the tree closest to the root are processed in sub-saturated regime, i.e. there is not enough parallelism in the application to run all the TCUs. The next $\log(N/p) - 1$ levels have more parallelism than the hardware can execute in parallel and some TCUs will run more than one thread (saturated regime). The computation starts at the first level above the leaves.

$$Execution\ Depth = (2\log_k N + 1) \times \mathcal{R} + (3k + 9)\log_k N + 2k + 33 \quad (3.5)$$

$$Additional\ Work = \frac{2N + (3k+2)Nodes_{sat}}{p} + (3k+2)\log_k p + \left\lceil \frac{Nodes_{sat}}{p} - \log_k \frac{N}{p} \right\rceil \times \mathcal{R} \quad (3.6)$$

To avoid starting too many short threads, the thread clustering optimization reviewed in Section 3.1.6.1 can be applied either by an optimizing compiler or a performance programmer. Let c be a constant; start c threads that each run a serial summation algorithm on a contiguous sub-array of N/c values from the input array. Each thread writes its computed partial sum into an array B. To compute the total sum, run the parallel summation algorithm described above on the array B.

We now consider how clustering changes the execution time. $SerSum(N)$ and $ParSum(N)$ denote the execution time for serial and the parallel summation algorithms over N values respectively. We determined the constant factors below by examining the XMTC code provided in Appendix A.2. The serial algorithm loops over N elements and, by using prefetching to always have the next value available before it is needed, it has a running time of $SerSum(N) = 2N + 1 \times \mathcal{R}$.

The first part of the algorithm uses a total of $N - c$ additions evenly divided among p

processors, while the second requires the parallel summation to be applied on an input of size c. This gives an execution time for the clustered algorithm of:

$$Execution\ Time = SerSum(\frac{N-c}{p}) + ParSum(c) \tag{3.7}$$

The value of c, where $p \leq c \leq N$, that minimizes the execution time determines the best crossover point for clustering. Suppose $p = 1024$. To allow numerical comparison, we needed to assign a value to $\mathcal{R}$, the number of cycles in one RTM. As noted in Section 3.1.6, We assume a value of 24 cycles for $\mathcal{R}$, which was confirmed experimentally for the prototype XMT architecture under the assumption that the data is already in the on-chip cache and there is no queuing in the interconnection network or memory.

Since all the clustered threads are equal in length and in the XMT Execution Model they run at the same speed, we found, that when $N \gg p$, the optimal value for c is 1024.

The optimum value for k can be determined by minimizing execution time for a fixed N. For the interesting case when $N \geq p$ (where $p = 1024$), the parallel summation algorithm is only run on $c = 1024$ elements and in this case we empirically determined that $k = 8$ is optimal. We ran a similar study for determining the sensitivity to the k factor for the Prefix-Sums problem, discussed next. The results of that experiment are presented in Figure 3.8a. A similar result holds for the Parallel Summation problem.

3.3.2 Prefix-Sums

Computing the prefix-sums for n values is a basic routine underlying many parallel algorithms. Given an array $A[0..n-1]$ as input, let $prefix\_sum[j] = \sum_{i=0}^{j-1} A[i]$ for j between 1 and n and $prefix\_sum[0] = 0$. Two prefix-sums implementation approaches are presented and compared: The first algorithm considered is closely tied to the synchronous ("textbook") PRAM prefix-sums algorithm while the second one uses a no-busy-wait paradigm [Vis00]. The main purpose of the current section is to demonstrate designs of efficient XMT implementation and the reasoning that such a design may require. It is perhaps a strength of the modeling in our work that it provides a common platform for evaluating rather different algorithms. Interestingly enough, our analysis suggests that when it comes to addressing the most time consuming elements in the computation, they are actually quite similar.

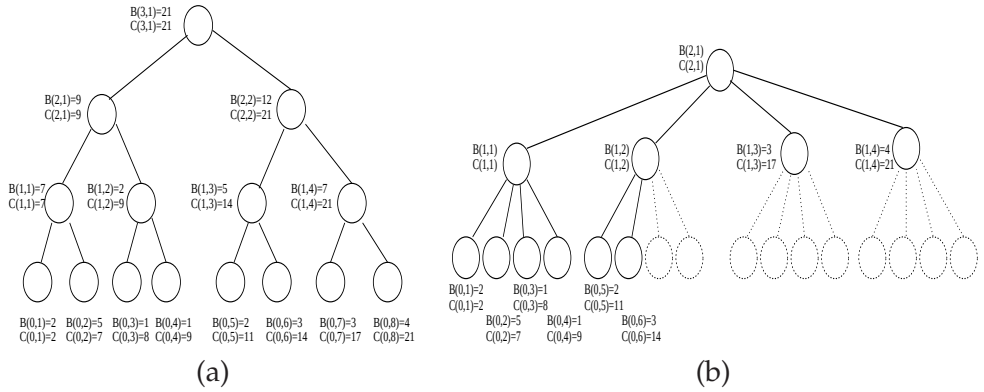


Figure 3.5: PRAM Prefix-Sums Algorithm Execution (a) Using a binary tree and (b) Using a k-ary tree (k=4).

Due to [LF80], the basic routine works in two stages each taking $O(\log n)$ time. The first stage is similar to the parallel summation algorithm presented in Section 3.3.1, namely the computation advances up a balanced tree computing sums. The second stage advances from root to leaves. Each internal node has a value $C(i)$, where $C(i)$ is the prefix-sum of its rightmost descendant leaf. The $C(i)$ value of the root is the sum computed in the first stage, and the $C(i)$ for other nodes is computed recursively. Assuming that the tree is binary, any right child inherits the $C(i)$ value from its parent, and any left child takes $C(i)$ equal to the $C(i)$ of its left uncle plus this child's value of sum. The values of $C(i)$ for the leaves are the desired prefix-sums. See Figure 3.5.a.

3.3.2.1 Synchronous Prefix-Sums

A pseudo-code implementation of this algorithm in the XMT Programming model is included in Figure 3.6. A full XMTC implementation is attached in Appendix A.3. Similar to the Summation algorithm, we used a k-ary tree instead of a binary one. The two overlapped k-ary trees are stored using two one-dimensional arrays sum and $prefix\_sum$ by using the array representation of a complete tree as discussed in Section 3.3.1.

The XMT algorithm works by first advancing up the tree using a summation algorithm. Then the algorithm advances down the tree to fill in the array $prefix\_sum$. The value of $prefix\_sum$ is defined as follows: (a) for a leaf, $prefix\_sum$ is the prefix-sum and (b) for an internal node, $prefix\_sum$ is the prefix-sum for its leftmost descendant leaf (not the rightmost descendant leaf as in the PRAM algorithm - this is a small detail that turns out to make things easier for generalizing from binary to k-ary trees).

```
/* Input: N numbers in sum[0..N-1]
 * Output: the prefix-sums of the numbers in
 *  prefix_sum[offset_to_1st_leaf..offset_to_1st_leaf+N-1]
 * The prefix_sum array is a 1D complete tree
 * representation (See Summation) */
kary_tree_summation(sum); // run k-ary tree summation algorithm
prefix_sum[0] = 0; level = log_k(N);
while(level > 0) { // all levels from root to leaves
    spawn(current_level_start_index, current_level_end_index) {
        int count, local_ps = prefix_sum[$];
        for(count = 0; count < k; count++) {
            prefix_sum[k*$ + count + 1] = local_ps;
            local_ps += sum[k*$ + count + 1]; }
    }
    level--;
    compute current_level_start and end_index
}
```

Figure 3.6: XMTC pseudo-code for k-ary Tree Prefix-Sums

Analysis of Synchronous Prefix-Sums Let us analyze the algorithm in the XMT Execution Model by examining the code in Appendix A.3. The algorithm has 2 round-trips to memory for each level going up the tree. One is to read sum from the children of a node, done in one RTM by prefetching all needed values in one round-trip. The other is to join the spawn at the current level. Symmetrically, there are 2 RTMs for each level going down the tree. One to read $prefix\_sum$ of the parent and sum of all the children of a node. Another to join the spawn at the current level. This gives a total of $4 * \log_k N$ RTMs. There is no queuing.

In addition to RTMs, there is a computation cost. The depth is $O(\log_k N)$ due to ascending and descending a logarithmic depth tree. By analyzing the XMTC implementation in Appendix A.3 we observed this term to be $(7k + 18)\log_k N + 2k + 39$ portion of the depth formula. We determined the $Additional\ Work$ similarly to the summation algorithm. It contains a $\frac{3N}{p}$ term for copying data to the tree's leaves and a $\frac{C*(N-min(p,N-1)-1)/(k-1)}{p} + C * \log_k p$ term to advance up and down the tree. This is derived by using the geometric series to count the number of internal nodes in the tree (because each internal node is touched by one thread and $C = (7k + 4)$ is the work per node) and considering that processing any level of the tree with fewer than p nodes has $Additional\ Work = \frac{(C \times p_i)}{p_i} = C$. The overhead to start threads in over-saturated conditions is computed analogously.

For the moment, we are not considering how clustering could be applied. Assuming that a round-trip to memory takes $\mathcal{R}$ cycles, the performance of this implementation is as follows:

$$Execution\ Depth = (4\log_k N + 3) \times \mathcal{R} + (7k + 18)\log_k N + 2k + 39 \quad (3.8)$$

$$\begin{aligned} Additional\ Work = {} & \frac{3N + (7k+4)(N - min(p, N-1) - 1)/(k-1)}{p} + \\ & +(7k+4)\log_k p + \\ & \left\lceil \frac{(N - min(p, N-1) - 1)/(k-1)}{p} - log_k \frac{N}{p} \right\rceil \times 2\mathcal{R} \end{aligned} \quad (3.9)$$

3.3.2.2 No-Busy-Wait Prefix-Sums

A less-synchronous XMT algorithm is presented. The Synchronous algorithm presented above processes each level of the tree before moving to the next, but this algorithm has no such restriction. The algorithm is based on the No-Busy-Wait balanced tree paradigm [Vis00]. As before, we used k-ary rather than binary trees.

The input and data structures are the same as previously, with the addition of the $gatekeeper$ array, providing a "gatekeeper" variable per tree node. The computation advances up the tree using a No-Busy-Wait summation algorithm. Then it advances down the tree using a No-Busy-Wait algorithm to fill in the prefix-sums.

The pseudo-code of the algorithm in the XMT Programming Model is included in Figure 3.7. A full XMTC implementation is included in Appendix A.4.

Analysis of No-Busy-Wait Prefix-Sums We determined the following factors by examining the XMTC implementation included in Appendix A.4. When climbing the tree, the implementation executes 2 RTMs per level, just as in the previous algorithm. One RTM is to read values of sum from the children, and the other is to use an atomic Prefix-sum instruction on the gatekeeper. The LSRTM to descend the tree is also 2 RTMs per level. First, a thread reads the thread ID assigned to it by the parent thread, in one RTM. The second RTM is used to read $prefix\_sum$ from the parent and sum from the children in order to do the necessary calculations. This is an LSRTM of $4\log_k N$. Also, there are additional $O(1)$ RTMs.

```
Spawn(first_leaf,last_leaf)
    Do while alive
       Perform psm on parent gatekeeper
       If last to arrive at parent
         Move to parent and sum values from children
       Else
         Join
    If at root
          Join

prefix_sum[0] = 0          //set prefix_sum of root to 0
Spawn(1,1)              //spawn one thread at the root
    Let prefix_sum value of left child = prefix_sum of parent
    Proceed through children left to right where each child is
    assigned prefix_sum value equal to prefix_sum + sum of left
    sibling      Use a nested spawn command to start a thread to
    recursively handle each child thread except the leftmost.
    Advance to leftmost child and repeat.
```

Figure 3.7: XMTC pseudo-code for k-ary No-Busy-Wait Prefix-Sums

Queuing is also a factor. In the current algorithm, up to k threads can perform a prefix-sum-to-memory operation concurrently on the same gatekeeper and create a $k-1$ queuing delay (since the first access does not count towards queuing delay). The total QD on the critical path is $(k-1)\log_k N$.

In addition to RTMs and QD, we counted computation depth and work. The computation depth is $O(\log_k N)$. Counting the constants in the XMTC implementation yields $(11+8k)*\log_k N + 2k + 55$.

The $\Sigma\frac{Work}{min(p,p_i)}$ part of the complexity is derived similarly as in the synchronous algorithm. It contains an $\frac{18N}{p}$ term, which is due to copying data to the tree's leaves and also for some additional work at the leaves. There is a $\frac{C*(N-min(p,N-1)-1)/(k-1)}{p} + C*\log_k p$ term to traverse the tree both up and down. This value is derived by using the geometric series to count the number of internal nodes in the tree and multiplying by the work per internal node ($C = (11+8k)$) as well as considering that processing any level of the tree with fewer than p nodes has $\frac{Work}{min(p,p_i)} = C$. Without considering clustering, the running time is given by:

$$
\begin{aligned}
Execution\ Depth &= (4\log_k N + 6) \times \mathcal{R} + (11 + 9k) * \log_k N + 2k + 55 && (3.10)\\
Additional\ Work &= \frac{6 + 18N + (11 + 8k)(N - min(p, N-1) - 1)/(k-1)}{p} + \\
&\quad +(11 + 8k)\log_k p + \\
&\quad + \left\lceil \frac{(N - min(p, N-1) - 1)/(k-1)}{p} - log_k \frac{N}{p} \right\rceil \times 2\mathcal{R} && (3.11)
\end{aligned}
$$

3.3.2.3 Clustering for Prefix-sums

Clustering may be added to the Synchronous k-ary prefix-sums algorithm to produce the following algorithm. The algorithm begins with an embarrassingly parallel section, uses the parallel prefix-sums algorithm to combine results, and ends with another embarrassingly parallel section.

1. Let c be a constant.
   Spawn c threads that run the serial summation algorithm on a contiguous sub−array of N/c values from the input array. The threads write the resulting sum values into a temporary array B.
2. Invoke the parallel prefix−sums algorithm on array B.
3. Spawn c threads. Each thread retrieves a prefix−sum value from B. The thread then executes the serial prefix−sum algorithm on the appropriate sub−array of N/c values from the original array.

The No-Busy-Wait prefix-sums algorithm can be clustered in the same way.

We now present the formulas for execution time using clustering. Let c be the number of threads that are spawned in the embarrassingly parallel portion of the algorithm. Let $SerSum$ be the complexity of the serial summation algorithm, $SerPS$ be the complexity of the serial PS algorithm, and $ParPS$ be the complexity of the parallel PS algorithm (dependent on whether the synchronous or No-Busy-Wait is used).

We examined the serial summation and serial prefix-sum implementations included in Appendixes A.2 and A.5 respectively to determine the following quantities. The serial sum and prefix-sum algorithms loop over N elements and from the serial code it is derived that $SerSum(N) = 2N + 1 \times \mathcal{R}$ and $SerPS(N) = 3N + 1 \times \mathcal{R}$.

The following formula calculates the cost of performing the serial algorithms on a set of $N-c$ elements divided evenly among p processors and then adds the cost of the parallel step:

$$Execution\ Depth = SerSum(\frac{N-c}{p}) + SerPS(\frac{N-c}{p}) + ParPS(c) \tag{3.12}$$

Optimal k and Optimal Parallel-Serial Crossover The value c, where $p \leq c \leq N$, that minimizes the formula determines the best crossover point for clustering. Let us say $p = 1024$ and $\mathcal{R} = 24$. Similar to the Summation problem, we concluded that in the XMT Execution Model for many values $N \geq p$, the best c is 1024. This is the case for both algorithms. A different value of c may be optimal for other applications, for example if the threads do not have equal work.

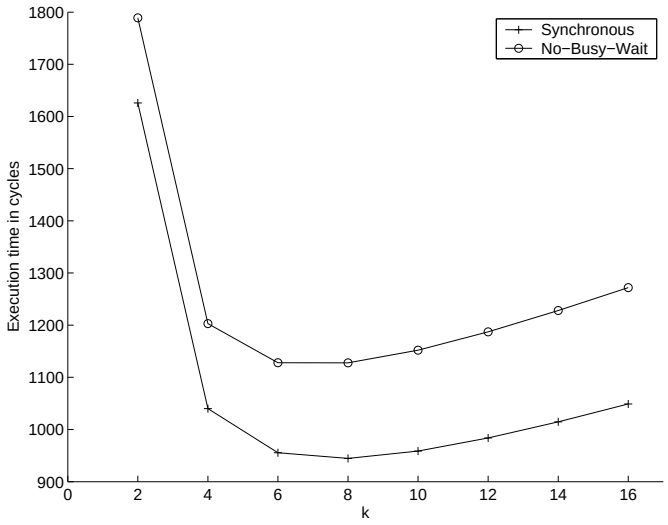
The optimal k value, where k denotes the arity of the tree, to use for either of the prefix-sums algorithms can be derived from the formulas. As shown in Figure 3.8a, for $N \geq p$ (where $p = 1024$), the parallel sums algorithm is only run on $c = 1024$ elements and in this case $k = 8$ is optimal for the synchronous algorithm and $k = 7$ is optimal for the No-Busy-Wait algorithm. When $N < p$, clustering does not take effect, and the optimal value of k varies with N, for both algorithms.

3.3.2.4 Comparing Prefix-sums Algorithms

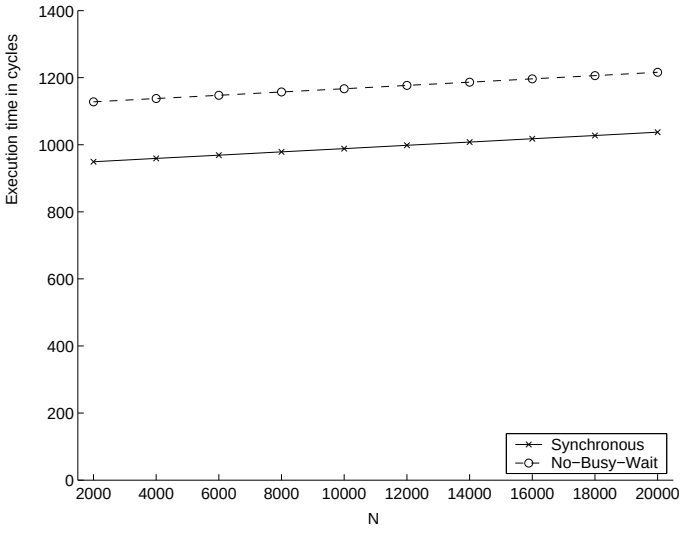
Using the performance model presented previously with these optimizations allows comparison of the programs in the XMT Execution Model. The execution time for various N was calculated for both prefix-sums algorithms using the formula with clustering. This is plotted in Figure 3.8b.

The Synchronous algorithm performs better, due to the smaller computation constants. The LSRTM of both algorithms is the same, indicating that using gatekeepers and nesting is equivalent in RTMs to using synchronous methods. The No-Busy-Wait algorithm has slightly longer computation depth and more computation work due to the extra overhead.

Note that in an actual XMT system, an implementation of the Prefix-Sums algorithm would be likely to be included as a library routine, relieving the programmer from the burden of implementing all the aforementioned optimizations.



(a) Determining the optimum arity of the tree k for the two implementations of the Prefix-Sums algorithm for $N = 1024$



(b) Execution times for the two implementations of the k-ary Prefix-Sums algorithms. The optimum k is chosen for each case

Figure 3.8: Estimated run times for Prefix-Sum obtained using the analytic model

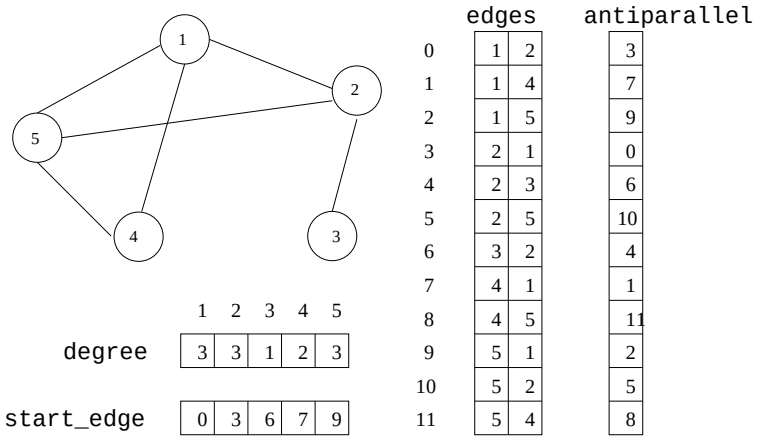


Figure 3.9: Example of the incidence list representation for a graph

3.3.3 Breadth-First Search

As noted earlier, Breadth-First Search (BFS) provides an interesting example for XMT programming. we are assuming that the graph is provided using the incidence list representation, as pictured in Figure 3.9.

Let $L(i)$ be the set of $N(i)$ nodes in level i and $E(i)$ the set of edges adjacent to these nodes. We illustrate only illustrate how to implement one iteration. Developing the full program based on this is straightforward.

As described in Section 3.1.3, the High-Level Work-Depth presentation of the algorithm starts with all the nodes in parallel, and then using nested parallelism ramps up more parallelism to traverse all their adjacent edges in one step. Depending on the extent that the target programming model supports nested parallelism, the programmer needs to consider different implementations. These choices are discussed below, laying out assumptions regarding the target XMT model.

We noted before that the Work-Depth model is not a direct match for our proposed programming model. With this in mind, we do not present a full Work-Depth description of the BFS algorithm; as will be shown, the "ideal" implementation will be closer to the High-Level Work-Depth presentation.

3.3.3.1 Nested Spawn BFS

In a XMT programming model that supports nested parallel sections, the High-level XMT program can be easily derived from the HLWD description:

```
For every vertex v of current layer L(i) spawn a thread
    For every edge e=(v,w) adjacent on v spawn a thread
        Traverse edge e
```

An XMTC pseudo-code implementation of this algorithm using the XMTC programming language is included in Figure 3.10. To traverse an edge, threads use an atomic prefix-sum instruction on a special "gatekeeper" memory location associated with the destination node. All gatekeepers are initially set to 0. Receiving a 0 from the prefix-sum instruction means the thread was the first to reach the destination node. The newly discovered neighbors are added to layer $L(i+1)$ using another prefix-sum operation on the size of $L(i+1)$. In addition, the edge anti-parallel to the one traversed is marked to avoid needlessly traversing it again (in the opposite direction) in later BFS layers. Note that this matches the PRAM Arbitrary Concurrent Write convention, discussed in Section 2.1.

The Nested Spawn algorithm bears a natural resemblance to the HLWD presentation of the BFS algorithm and in this sense, is the ideal algorithm to program. Allowing this type of implementations to be written and efficiently executed is the desired goal of a XMT framework.

Several other XMT BFS algorithms will be presented to demonstrate how BFS could be programmed depending on the quantitative and qualitative characteristics of a XMT implementation.

3.3.3.2 Flattened BFS

In this algorithm, the total amount of work to process one layer (i.e. the number of edges adjacent to its vertices) is computed, and it is evenly divided among a pre-determined number of threads p, value which depends on architecture parameters. For this, a Prefix-sums subroutine is used to allocate an array of size $|E(i)|$. The edges will be laid out flat in this array, located contiguously by source vertex. p threads are then spawned, each being assigned one sub-array of $|E(i)|/p$ edges and traversing these edges one by one. An illustration of the steps in this algorithm can be found in Figure 3.11.

To identify the edges in each sub-array, it is sufficient to find the first (called *marker*) edge in such an interval; one can then use the natural order of the vertices and edges to find the rest. The algorithm starts by identifying first (if any) marker edge adjacent to v_j

```
/* Input: Graph G=(E,V) using adjacency lists
 * Output: distance[N] - distance from start for each vertex
 * Uses: level[L][N] - sets of vertices at each BFS level */

/* run prefix sums on degrees to determine position of start
edge for each vertex */
start_edge = kary_prefix_sums(degrees);
level[0]=start_node; i=0;
while (level[i] not empty) {
   /* start one thread for each vertex in level[i] */
   spawn(0,level_size[i] - 1) {
      v = level[i][$]; // read one vertex
      /* start one thread for each edge of each vertex */
      spawn(0,degree[v]-1) {
          /* read one edge (v,w) */
          int w = edges[start_edge[v]+$][2];
          /* check the gatekeeper of the end-vertex w */
          psm(gatekeeper[w],1);
          if gakeeper[w] was 0 {
             /* allocate one entry in level[i+1] */
             ps(current_level_size,1);
             store w in level[i+1];
          }
      } /* join */
   }   /* join */
   i++;
}
```

Figure 3.10: XMTC pseudo-code for Nested Spawn Breadth-First Search

for all vertices $v_j \in L(i)$ in parallel, then use a variant of the pointer jumping technique [JáJ92, Vis07] to identify the rest of the marker edges (if any) adjacent to v_j using at most $\log_2 p$ steps.

Flattened BFS Analysis When each of p threads traverses the edges in its sub-array serially, a simple optimization would prefetch the next edge data from memory and overlap the prefix-sum operations, thus reducing the number of round-trips to memory from $O(\frac{|E(i)|}{p})$ to a small constant. Such an improvement can be quite significant.

The Flattened BFS algorithm uses the prefix sums algorithm as a procedure; we used the running time computed for this routine in Section 3.3.2.

A full XMTC implementation of the Flattened BFS algorithm is included in Appendix

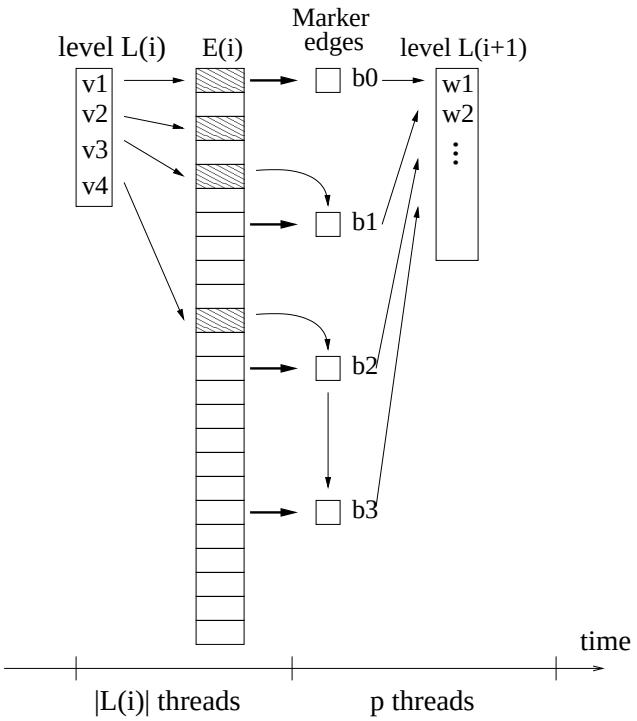


Figure 3.11: Execution of Flattened BFS algorithm. First allocate $E[i]$ to hold all edges adjacent to $level[i]$. Next, identify marker edges b_i, which give the first edge per each sub-array. Running one thread per sub-array, all edges are traversed to build $level[i+1]$

A.6. By analyzing this implementation, we determined that identifying the marker edges uses 3 RTMs to initialize one marker edge per vertex, and then $4 \log_2 p$ RTMs to do $\log_2 p$ rounds of pointer jumping and find the rest of the adjacent marker edges. Finally, p threads cycle through their sub-arrays of $\frac{|E(i)|}{p}$ edges.

By using the optimizations described above, the only roundtrip to memory penalties paid in this step are that of traversing a single edge. A queuing delay occurs at the node gatekeeper level if several threads reach the same node simultaneously. This delay depends on the structure of the graph, and is denoted GQD in the formula below.

In addition to LSRTM and QD, the computation depth also appears in the depth formula. The $10 \log_2 p$ term is the computation depth of the binary tree approach to identifying marker edges. The computation depth of the call to prefix-sums is included.

The dominating term of the $Additional\ Work$ is $7|E(i)|/p + 28N(i)/p$, which comes from the step at the end of the algorithm in which all the edges are traversed in parallel by p threads, and the new found vertices are added to level $L(i+1)$. The $Additional\ Work$ portion of the complexity also contains the work for the call to prefix-sums. The perfor-

mance is:

$$\begin{aligned} Execution\ Depth &= (4\log_k N(i) + 4\log_2 p + 14) \times \mathcal{R} + 38 + 10\log_2 p + 7|E(i)/p| + \\ &\quad +16N(i)/p + GQD + Computation\ Depth(Prefix\ sums) \quad (3.13) \\ Additional\ Work &= \frac{7|E(i)| + 28N(i) + 15p + 17}{p} + \lceil \frac{N(i) - p}{p} \rceil \times \mathcal{R} + \\ &\quad +Additional\ Work(Prefix\ sums) \quad (3.14) \end{aligned}$$

As before, $N(i)$ is the number of nodes in layer $L(i)$, $|E(i)|$ is the number of edges adjacent to $L(i)$ and $\mathcal{R}$ is the number of cycles in one RTM.

The second term of relation 3.14 denotes the overhead of starting additional threads in over-saturated conditions. In the flattened algorithm, this can occur only in the initial phase, when the set of edges $E(i)$ is filled in. To reduce this overhead, we can apply clustering to the relevant parallel sections.

Note the following special case: when the number of edges adjacent to one layer is relatively small, there is no need to start p threads to traverse them. We choose a threshold θ, and if $\frac{|E(i)|}{p} < \theta$, then we use $p' = \frac{|E(i)|}{\theta}$ threads. Each will process θ edges. In this case, the running time is found by taking the formulas above and replacing p with $p' = \frac{|E(i)|}{\theta}$.

3.3.3.3 Single-spawn and k-spawn BFS

Although the programming model can allow nested parallelism, the execution model might include limited or no support for nesting. To provide insight into the transformations applied by the compiler, and how to reason about the efficiency of the execution, We present two implementations of the Nested Spawn BFS algorithm that directly map into an Execution model with limited support for nesting.

Single-Spawn Algorithm and Analysis The single-spawn BFS Algorithm uses `sspawn()` and a binary tree type technique to allow the nested spawning of any number T of threads in $\log_2 T$ steps. The algorithm spawns one thread for each vertex in the current level, and then uses each thread to start $degree(vertex) - 1$ additional threads by iteratively using the `sspawn()` instruction to delegate half a thread's work to a new thread. When one edge per thread is reached, the edges are traversed.

The pseudo-code for a single layer is as follows.

```
For every vertex v of current layer L spawn a thread
    While a thread needs to handle s > 1 edges
        Use sspawn() to start another thread and
          delegate floor(s/2) edges to it
    Traverse one edge
```

In this algorithm, one thread per each vertex v_i is started, and each of these threads then repeatedly uses single-spawn to get $deg(v_i) - 1$ threads started.

To estimate the running time of this algorithm, we enumerate the operations that take place on the critical path during the execution. The constant factors are determined by examining the full implementation of the Single-Spawn BFS included in Appendix A.7.

- Start and initialize the original set of $N(i)$ threads, which in our implementation takes 3 RTMs to read the vertex data.
- Let d_{max} be the largest degree among the nodes in current layer. Use single-spawn and $\log_2 d_{max}$ iterations to start d_{max} threads using a balanced binary tree approach. Starting a new thread at each iteration takes 2 RTMs (as described in section 3.1.6), summing up $2 \log_2 d_{max}$ RTM on the critical path.
- The final step of traversing edges implies using one prefix-sum instruction on the gatekeeper location and another one to add the vertex to the new layer.

The cost of queuing at gatekeepers is represented by GQD. In my implementation, the computation depth was $18 + 7 \log_2 d_{max}$.

Up to $|E(i)|$ threads are started using a binary tree, and when this number exceeds the number of TCUs p, we account for the additional work and the thread starting overhead. We estimated these delays by following the same reasoning as with the k-ary Summation algorithm in Section 3.3.1 using a constant of $C = 19$ cycles work per node as implied by our implementation.

The performance is:

$$Execution\ Depth = (7 + 2\log_2 d_{max})\mathcal{R} + (18 + 7\log_2 d_{max}) + GQD \quad (3.15)$$

$$Additional\ Work = \frac{19(|E(i)| - min(p, |E(i)| - 1) - 1) + 2}{p} + \\ +19\log_2 |E(i)| + \left\lceil \frac{|E(i)| - p}{p} \right\rceil \times \mathcal{R} \quad (3.16)$$

To avoid starting too many threads, the clustering technique presented in Section 3.1.6.1 can be applied. This will reduce the additional work component since the cost of allocating new threads to TCUs will no longer be paid for every edge.

k-spawn Algorithm and Analysis The k-spawn BFS Algorithm follows the same principle as Single-spawn BFS, but uses the `kspawn()` instruction to start the threads faster. By using a k-ary rather than binary tree to emulate the nesting, the number of steps to start T threads is reduced to $\log_k T$.

The k-spawn BFS pseudo-code for processing one layer is:

```
For every vertex v of current layer L spawn a thread
    While a thread needs to handle s > k edges
        Use kspawn() to spawn k threads and
          delegate to each floor(s/(k+1)) edges
    Traverse (at most) k edges
```

The threads are now started using k-ary trees and are therefore shorter. The LSRTM is $2\log_k d_{max}$. The factor of 2 is due to the 2 RTMs per k-spawn.

The full XMTC implementation of the k-spawn BFS algorithm is included in Appendix A.8. By examining this implementation, We determined the computation depth to be $(5 + 4k)\log_k d_{max}$. This is an $O(k)$ cost per node, where $\log_k d_{max}$ nodes are on the critical path. The queuing cost at the gatekeepers is denoted by GQD. The $Additional\ Work$ is computed as in Single-spawn BFS with the constant $C = 4k + 3$ denoting the work per node in the k-ary tree used to spawn the $|E(i)|$ threads.

The performance is:

$$Execution\ Depth = (7 + 2\log_k d_{max})\mathcal{R} + (5 + 4k)\log_k d_{max} + 15 + 4k + GQD \quad (3.17)$$

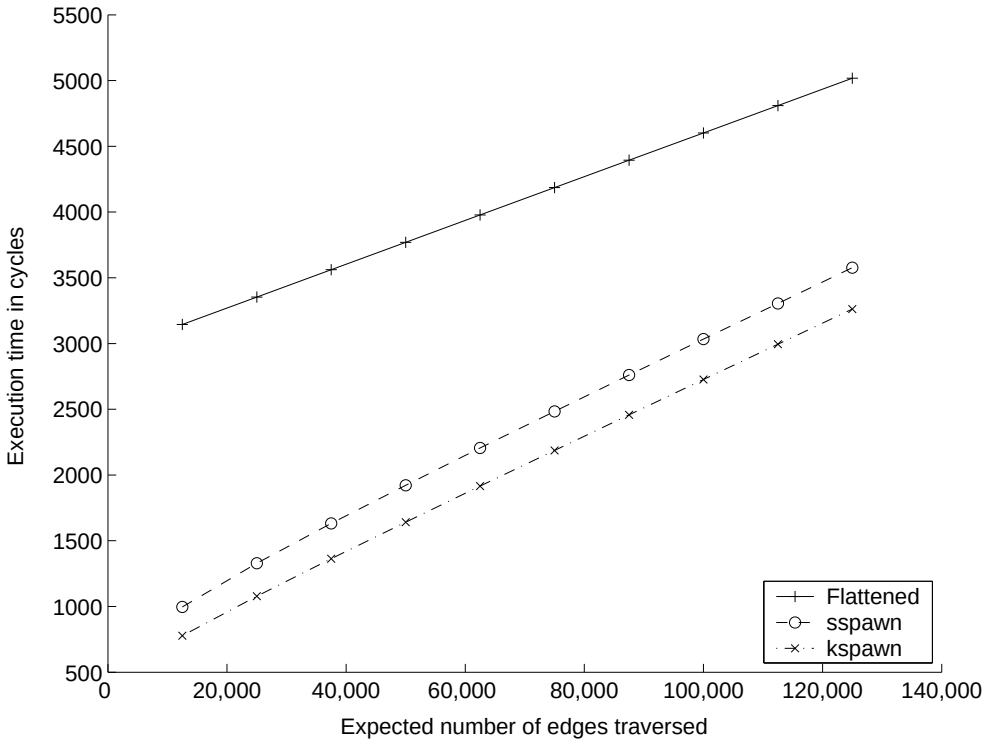
$$Additional\ Work = \frac{14|E(i)| + (4k+3)(|E(i)| - min(p, |E(i)| - 1) - 1)/(k-1)}{p} + \\ +(4k+3)\log_k |E(i)| + \left\lceil \frac{|E(i)| - p}{p} \right\rceil \times \mathcal{R} \quad (3.18)$$

Similar to the case of the Single-spawn BFS algorithm, thread clustering can be used in the k-spawn BFS algorithm by checking the number of virtual threads to determine whether the k-spawn instruction should continue to be used or if additional spawning is to be temporarily suspended.

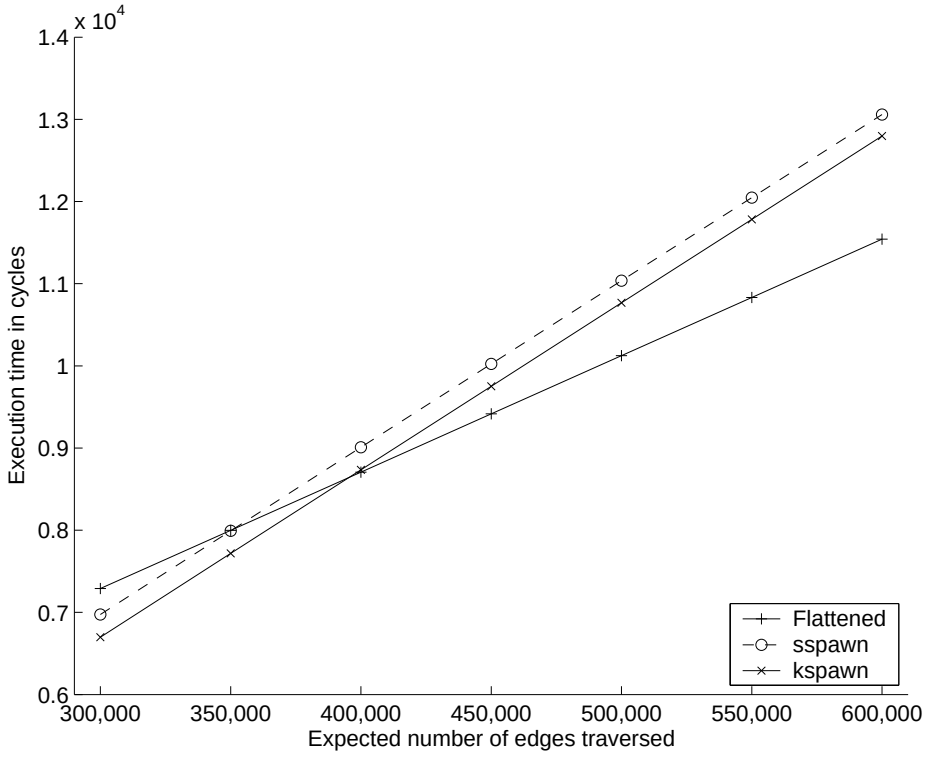
3.3.3.4 Comparing BFS Algorithms

Using the results of the analysis above, we calculated execution time for one iteration (i.e., processing one BFS level) of the BFS algorithms presented here and the results are depicted in Figure 3.12. This was done for two values for the number of vertices $N(i)$ in current level $L(i)$, 500 and 2000. The analysis assumes that all edges with one end in $L(i)$ lead to vertices which have not been visited in a previous iteration; since there is more work to be done for a "fresh" vertex, this constitutes a worst-case analysis. The same graphs are also used in Section 3.4 for empirically computing speedups over serial code as it is unlikely they significantly favor a parallel program over the serial one. To generate the graphs, we picked a value M and choose the degrees of the vertices uniformly at random from the range $[M/2, 3M/2]$, which gives a total number of edges traversed of $|E(i)| = M * N(i)$ on average. Only the total number of edges is shown in Figure 3.12. We arbitrarily set $N(i+1) = N(i)$, which gives a queuing delay at the gatekeepers (GQD) of $\frac{|E(i)|}{N(i)} = M$ on average. As stated in Section 3.1.6, we use the value $\mathcal{R} = 24$ for the number of cycles needed for one roundtrip to memory.

For small problems, the k-spawn algorithm came ahead and the Single-spawn one was second best. For large problems, the Flattened algorithm performs best, followed by k-spawn and Single-spawn. When the hardware is sub-saturated, the k-spawn and Single-spawn algorithms do best because their depth component is short. These algorithms have an advantage on smaller problem sizes due to their lower constant factors. The k-



(a) $N(i) = 500$



(b) $N(i) = 2000$

Figure 3.12: Analytic execution times for one iteration of BFS when the number of vertices at current level is 500 respectively 2000. The optimal value for k was calculated for each dataset

spawn implementation performs better than Single-spawn due to the reduced height of the "Spawn tree". The Flattened algorithm has a larger constant factor for the number of RTMs, mostly due to the introduction of a setup phase which builds and partitions the array of edges. For super-saturated situations, the Flattened algorithm does best due to a smaller work component than the other algorithms.

Note that using the formulas ignores possible gaps in parallel hardware usage. In a highly unbalanced graph, some nodes have high degree while others have low degree. As many nodes with small degree finish, it may take time before the use of parallel hardware can be ramped up again. For example, in the Single-spawn and k-spawn algorithms, the virtual threads from the small trees can happen to take up all the physical TCUs and prevent the deep tree from getting started. The small trees may all finish before the deep one starts. This means we are paying the work of doing the small trees plus the depth of the deep tree. A possible workaround would be to label threads according to the amount of work they need to accomplish and giving threads with more work a higher priority (e.g. by scheduling them to start as soon as possible). Note that this issue does not affect the Flattened BFS algorithm, since the edges in a layer are evenly distributed among the running threads.

3.4 Empirical Validation of the Performance Model

This section presents some empirical validation of the analytic performance model introduced in this chapter. Given an XMTC program, we compared estimated run-times using the analytic model with simulated run-times using XMTSim, our XMT cycle-accurate simulator.

A gap between the simulations and the analytic model is to be expected. The analytic model as presented makes some simplifying assumptions, such as counting each XMTC statement as one cycle, and ignoring contention at the functional units. It also provides the same (worst-case) run-time estimates for different input data as long as the input has the same size.

All these factors could cause the simulated run-times to be different than the ones computed by the analytic approach. For that reason, we are limiting the focus of this work to just studying the relative performance of two or more implementations that solve the

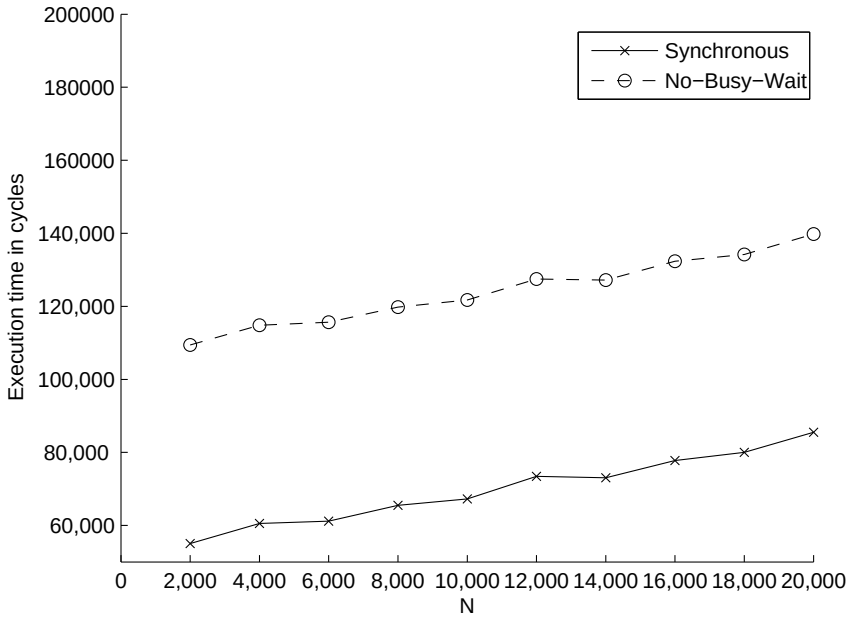


Figure 3.13: Cycle counts reported by the XMTSim cycle-accurate simulator. Synchronous and No-Busy-Wait Prefix-Sums

same problem. This enables evaluating programming implementation approaches against the relative performance gain, the main goal of the analytical performance model.

We simulated the implementations of the Prefix-Sums [LF80, Vis00] and the parallel Breadth-First Search algorithms discussed in Sections 3.3.2 and 3.3.3 using XMTSim.

Figure 3.13 presents the results reported for k-ary prefix-sums ($k = 2$) by the XMT cycle-accurate simulator for input sizes in the range $2,000 \ldots 20,000$. The results show the synchronous program outperforming the no-busy-wait program and the execution times increasing linearly with N. This is in agreement with the analytic model results in Figure 3.12.

However, there are some differences, as well. For example, the larger gap that the simulator shows between the synchronous program and no-busy-wait execution times. This can be explained by the relatively large overheads of the single-spawn instruction, which can be mitigated in the future by more efficient hardware implementation. Alternatively, once compiler support matures, a nested spawn implementation can be implemented, which will likely provide the best trade-off between development time and performance.

Figure 3.14 depicts the number of cycles obtained by running two XMTC implementa-

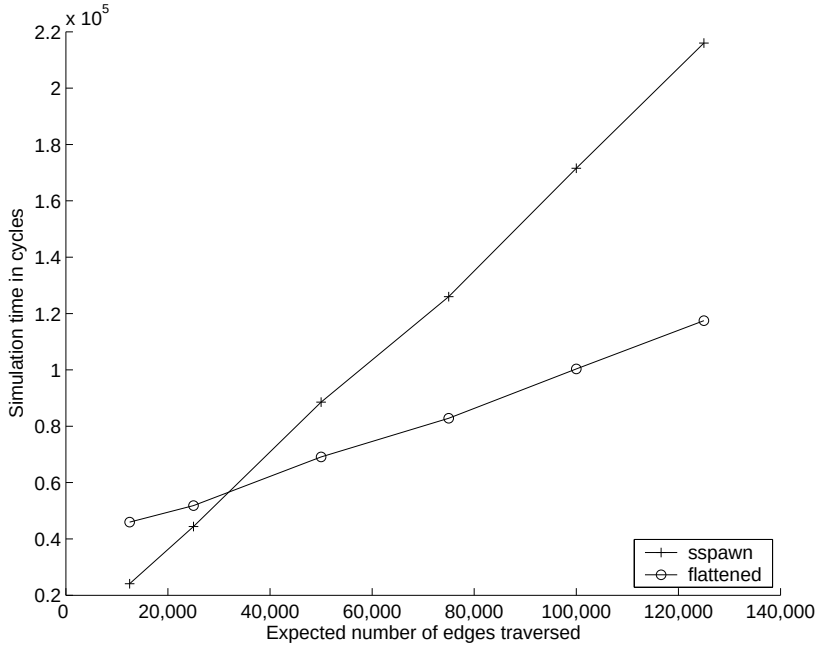


Figure 3.14: Cycle counts reported by the XMT cycle-accurate simulator. Flattened and Single-Spawn BFS

tions of the Breadth First Search algorithm, Single-Spawn and Flattened, on the simulator. The results show the execution times for only one iteration of the BFS algorithm, i.e., building BFS tree level $L(i+1)$ given level $L(i)$. To generate the graphs, we picked a value M and choose the degrees of the vertices uniformly at random from the range $[M/2, 3M/2]$, which gives a total number of edges traversed of $|E(i)| = M * N(i)$ on average. Only the total number of edges is shown in Figure 3.14. Another arbitrary choice was to set $N(i+1) = N(i)$, which gives gatekeeper queuing delay (GQD) of $\frac{|E(i)|}{N(i)} = M$ on average.

The number of vertices in levels $L(i)$ and $L(i+1)$ was set to 500. By varying the average degree per vertex M, we generated graphs with the expected number of edges between $L(i)$ and $L(i+1)$ in the range $[12,500..125,000]$. We observed that the single-spawn implementation outperforms the Flattened BFS for smaller problem sizes, but the smaller work factor makes the latter run faster when the number of edges traversed increases above a certain threshold.

By comparing these experimental results with the outcome of the analysis presented in Figure 3.12, we observed the same performance ranking between the different implementations providing a second example where the outcome of the analytic performance model is consistent with the cycle-accurate simulations.

3.5 Related Work

Most other researchers that worked on performance modeling of parallel algorithms were concerned with other platforms. They focused on different factors than us. Helman and JáJá [HJ99] measured the complexity of algorithms running on SMPs using the triplet of maximum number of non-contiguous accesses by any processor to main memory, number of barrier synchronizations, and local computation cost. Their focus on non-contiguous accesses and barriers is sensible given the slow memory access and costly synchronization on SMPs. These quantities are less important in a PRAM-like environment.

Bader, Cong, and Feo [BCF05] found that in some experiments on the Cray MTA, the costs of non-contiguous memory access and barrier synchronization were reduced almost to zero by multi-threading and that performance was best modeled by computation alone. For the latest generation of the MTA architecture, a calculator for performance that includes the parameters of count of trips to memory, number of instructions, and number of accesses to local memory [FHKK05] was developed. The RTMs that we count are round trips to the shared cache, which is quite different, as well as queuing at the shared cache. Another significant difference is that we considered the effect of optimizations such as prefetch and thread clustering. Nevertheless, the calculator should provide an interesting basis for comparison between performance of applications on MTA and XMT.

The incorporation of queuing follows the well-known QRQW PRAM model of Gibbons, Matias and Ramachandran [GMR98]. A succinct way to summarize the modeling contribution of this thesis is that unlike previous practice QRQW becomes secondary, though still quite important, to LSRTM.

Chapter 4

Resource-Aware Prefetching for XMT

Ease of programming is a necessary condition for the success of a general-purpose many-core platform, and it is one of the main objectives of XMT. The programmer's workflow introduced in Chapter 3 provides the framework to assist programmers and system designers towards reaching this goal. In particular, targeting the programming model and not the execution model significantly lowers the programmer's effort, allowing them to focus on designing efficient parallel algorithms instead of on low-level details. At the same time, this indicates the need for capable system software (compiler and run-time libraries) that can effectively transform the resulting program for the execution model. One such transformation implies relying on compiler prefetching to optimize for the length of sequence of round-trips to memory (LSRTM).

This chapter proposes an enhanced compiler loop prefetching algorithm and compare it with existing solutions. We show that by providing the compiler with accurate information about the hardware and using this information in the loop prefetching pass, running time can be improved on average by up to 36.7% when compared to using a hardware-oblivious prefetch algorithm, depending on the hardware configuration. The resulting Resource-Aware Prefetch (RAP) algorithm is particularly beneficial for many-core architectures with limited per-core resources such as XMT.

4.1 Background and Motivation

Memory systems have been under a lot of pressure to keep up with the increasing demand for parallelism coming from every new generation of microprocessors. Super-scalar, out-of-order processors can have a large number of memory operations in flight in the execution window at one time. In simultaneous multi-threading (SMT) architectures, as well as multicores and manycores, the demand for Memory-Level Parallelism (MLP) has further increased. This has put additional pressure for memory systems to support numerous concurrent memory requests.

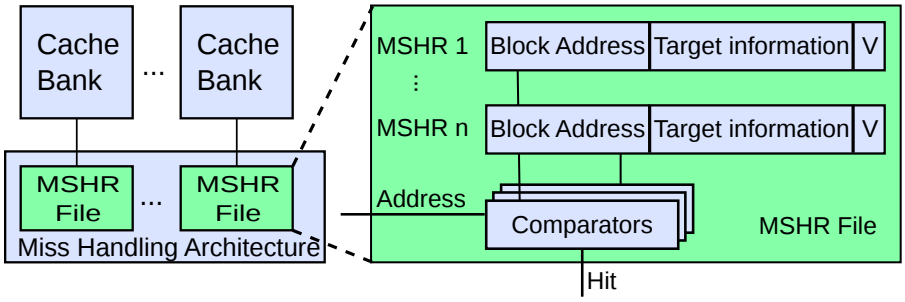


Figure 4.1: Miss Handling Architecture (MHA) for a banked cache system and the Miss Information/Status Holding Register (MSHR) file

Current cache hierarchy designs support only limited amounts of MLP due to the high cost in terms of chip area and energy use. Existing architectures employ lock-up free caches (e.g. [TCT06]) to avoid stalling the CPU and allow the cache miss to be serviced in the background. Fig. 4.1 depicts a cache system and its attached Miss Handling Architectures (MHA). This consists of several *Miss Information/Status Holding Register* (MSHR) files, and is responsible for keeping track of the outstanding concurrent misses. To meet the demand for high bandwidth and low latency, each MSHR has its own comparator, and is a small *fully associative cache.* The maximum number of outstanding cache misses the system supports is limited by the number of MSHR entries.

When a new request is received, all the comparators must be activated in parallel in order to retrieve the corresponding entry in one clock cycle, leading to high power consumption and area requirements. This severely limits the size of the MSHR file that can be included, even for today's large transistor budgets. For example, the L1 cache of an Intel Pentium 4 processor supports only 8 outstanding misses [BBH+04]. For AMD Opteron and Intel Core i7 architectures, an empirical study showed that single thread performance does not improve past 7 concurrent memory requests, suggesting that the same limitation holds [PFML09].

In many-core designs, each lightweight core has much smaller area than a traditional core. In terms of prefetching resources, this has two repercussions: (i) the size (number of entries) of MSHR files is much more constrained in a many-core due to area constraints; and (ii) the total energy consumption of all MSHRs across cores in a processor is much higher than in a traditional processor due to the larger number of cores, which further limits the size of MSHRs. As a consequence it becomes crucial to carefully manage the use

of scarce MSHR resources for these architectures.

We present a new prefetching method called Resource-Aware Prefetching (RAP) to manage MSHR resources carefully. It is mainly beneficial in many-cores because of their limited MSHR file capacity. It can apply to traditional multi-cores as well, but given their larger number of MSHR entries and their much larger area and power budget for other latency tolerating techniques mentioned above, the run-time gains from RAP are likely to be very small, hence we do not discuss those further.

4.2 Existing Software Prefetching Methods

Mowry et. al [MLG92] introduced a compiler algorithm to insert prefetch instructions into scientific applications that operate on dense matrices. Consider the code in Figure 4.2 as our running example. Figure 4.2(a) shows the original program code. In the figure, assume that the matrices A, B and C contain double precision floating point elements (64 bits) and our hypothetical system has a cache line of 16 bytes; thus two doubles fit per cache line. Also, assume that the cache miss latency is $MissLatency = 50$ clock cycles. Note that this simplified model, assuming only one level of cache and a fixed cache miss latency, is widely used in prefetching literature; accurately modeling the cache memory hierarchy in the compiler is often too complex to be viable. Moreover, since the cache is usually a system-wide shared resource, it is impossible to model interference from external sources such as other running processes. Mowry's algorithm proceeds as described in Algorithm 4.1.

Mowry's algorithm as presented successfully filters out most unnecessary prefetch instructions and significantly reduces the instruction overheads. However, it does not take into consideration the number of in-flight memory requests supported by the hardware. The maximum number of prefetch requests active at any time can be computed using:

$$MaxRequests = NumRefs \times PrefDistance \tag{4.2}$$

where *NumRefs* represents the number of static references that require prefetching. Going back to the code in Fig. 4.2(c), $NumRefs = 3$ since the references to `A[i]`, `B[i]` and `C[i]` will cause a cache miss at each iteration and need prefetching, leading to $MaxRequests = 3 \times 3 = 9$. Suppose that our architecture has 6 registers in the MSHR file. After the first six

(a)

```
for (i=0;i<1000;i++)
  A[i] = B[i] + C[i];
```

(b)

```
for (i=0;i<994;i += 2 ) { /* Unrolled */
  A[i] = B[i] + C[i];
  A[i+1] = B[i+1] + C[i+1];
}
/* Last three iterations peeled */
for (i=994;i<1000;i++)
  A[i] = B[i] + C[i];
```

(c)

```
for (i=0;i<994;i += 2 ) {
  /* prefetch 3 iterations in advance */
  prefetch(A[i+6]);
  prefetch(B[i+6]);
  prefetch(C[i+6]);
  A[i] = B[i] + C[i];
  A[i+1] = B[i+1] + C[i+1];
}
for (i=994;i<1000;i++)
  A[i] = B[i] + C[i];
```

(d)

```
for (i=0;i<994;i += 2 ) {
  prefetch(A[i+6]);
  prefetch(B[i+6]);
  /* Does not prefetch C */
  A[i] = B[i] + C[i] ;
  A[i+1] = B[i+1] + C[i+1];
}
for (i=994;i<1000;i++)
  A[i] = B[i] + C[i];
```

(e)

```
for (i=0;i<996;i += 2 ) {
  /* prefetch 2 iterations in advance */
  prefetch(A[i+4]);
  prefetch(B[i+4]);
  prefetch(C[i+4]);
  A[i] = B[i] + C[i] ;
  A[i+1] = B[i+1] + C[i+1];
}
/* Last two iterations peeled */
for (i=996;i<1000;i++)
  A[i] = B[i] + C[i];
```

Figure 4.2: (a) Original code before loop prefetching (b) Loop unrolling and peeling to isolate likely cache misses (c) Code after Mowry's prefetching algorithm (*PrefDistance* $= 3$) (d) Code after applying GCC loop prefetching algorithm (prefetch slots=6) (e) Outcome of the RAP algorithm: *PrefDistance* lowered to 2.

Algorithm 4.1 Mowry's Loop Prefetching

I. *For each static affine array reference, use locality analysis to determine which dynamic accesses are likely to suffer cache misses and therefore should be prefetched. For the code in Fig. 4.2(a), one cache line can hold two array elements, and thus every second dynamic access for the* `A[i]`, `B[i]` *and* `C[i]` *references will be a cache miss and requires a prefetch instruction.*

II. *Isolate the predicted dynamic miss instances using loop-splitting techniques such as peeling, unrolling, and strip-mining. This avoids the overhead of adding conditional statements for prefetching to the loop bodies. This yields the code in Fig. 4.2(b), where the loop has been unrolled two-fold and the last 6 iterations have been pulled out in a separate loop.*

III. *Schedule prefetches the proper amount of time in advance using software pipelining (by using the computed necessary prefetch distance), where the computation of one or more iterations is overlapped with prefetches for a future iteration. The prefetch distance is computed so that all latency can be hidden completely, using the formula:*

$$PrefDistance = \left\lceil \frac{MissLatency}{IterationTime} \right\rceil \tag{4.1}$$

IterationTime is the estimated running time of the shortest path through the loop when software prefetching is enabled. Assume for example that IterationTime $= 20$ clock cycles (after unrolling), and thus PrefDistance $= \lceil 50/20 \rceil = 3$ iterations. The code in Fig. 4.2(c) contains the transformed code, where prefetches for the references to the A, B and C arrays have been inserted three iterations in advance.

prefetch requests have been issued, when the next request arrives at the MHA unit, one of the following can happen, depending on the hardware implementation:

1. The additional request is silently dropped, and nothing is sent to the lower levels of the memory hierarchy. This causes the program to slow down, since it incurs all the instruction overheads of prefetching, but none of the benefits – the cache miss was not avoided.

2. The MHA does not accept the prefetch request, stalling the issuing CPU until one MSHR becomes available (which happens when one request returns from DRAM or lower cache level). Stalling the CPU was exactly what prefetching was aiming to avoid, and thus the benefits of prefetching are again lost, leaving only the overheads.

To summarize, overflowing the MSHR file is detrimental in all cases, and needs to be addressed by all prefetching approaches. The code in Fig. 4.2(c), which is the outcome of Mowry's algorithm, fails to address this issue. We discuss proposed improvements next.

GCC (GNU Compiler Collection), a state-of-the art open source compiler which supports a wide range of architectures and programming languages, includes an implementation of Mowry's algorithm for loop prefetching. The GCC algorithm extends it further by introducing the notion of a platform-specific number of *PrefetchSlots*. This is used to limit the number of prefetches that can be in flight at the same time. As far as we know, GCC's method is the only software prefetching algorithm that attempts to limit the number of in-flight prefetches based on hardware limitations. After performing the same steps 1-2 as above, the GCC algorithm starts scheduling prefetches for all the references in program order. One prefetch instruction issued *PrefDistance* iterations in advance of the reference causes the number of available prefetch slots to be decremented by *PrefDistance*. Once not enough *PrefetchSlots* are left, it stops issuing prefetches for the remaining references.

For our running example, Fig. 4.2(d) shows the outcome of the GCC algorithm. Since the prefetch instructions for the `A[i]` and `B[i]` references use up all 6 available prefetch slots, no prefetch is issued for the `C[i]` reference. At runtime, this means a cache miss penalty will be encountered every iteration of the unrolled loop, significantly affecting its running time. Although the GCC algorithm in Fig. 4.2(d) addressed the MSHR file overflowing issue encountered by Mowry's original approach in Fig. 4.2(c), it comes short of the main goal of hiding the memory latency of all the memory references.

In the next section we discuss an enhanced prefetching algorithm, which aims at addressing the limitations of previous approaches and obtain better runtime on a series of benchmarks.

4.3 New Resource-Aware Prefetching Method

4.3.1 Intuition

The main contribution of this chapter is a new compiler prefetching algorithm – Resource-Aware Prefetching (RAP) – which improves upon Mowry's standard loop prefetching algorithm as well as the GCC implementation by using the very limited MHA resources more efficiently. Our algorithm robustly adapts to constrained resources and uses them to hide as much latency as possible. More concretely, we show that in situations where not enough prefetch slots are available to issue prefetch instructions for all references, it is more beneficial to decrease the prefetch distance and prefetch for as many references as

possible. By contrast, the GCC implementation uses a fixed prefetch distance and may prefetch fewer references.

Fig. 4.2(e) shows the outcome of the RAP algorithm applied to our example code. The prefetch distance has been lowered to two iterations, which allowed prefetches to be issued for all three references. As will be discussed below, with this transformation there will be only *one cache miss per three iterations*: once a cache miss is encountered, it gives enough time for all previously issued prefetch requests to complete, including current and next two iterations. By contrast, the GCC implementation encounters one miss per each iteration, which translates to three times more time spent in memory stalls.

4.3.2 Implementation

To formulate an algorithm for RAP, it is useful to understand the limitations of GCC's prefetcher. There is a subtle inconsistency in the way GCC schedules prefetching instructions: on one hand, the prefetch distance is computed assuming all memory latencies can be hidden through prefetching; on the other hand, under certain conditions, prefetch instructions for some references are not even issued, causing some references to be cache misses. This affects the iteration time, and therefore the prefetch distance should be adjusted accordingly: if each iteration takes longer, then prefetches can be issued fewer iterations in advance and still be able to hide the latency. However, GCC does not adjust the prefetch distance in these cases, *effectively using a flawed model for scheduling prefetches.*

Figure 4.2(d) shows an example of the suboptimal scheduling algorithm described above. To help understand the runtime behavior, we show the resulting dynamic cache trace in Figure 4.3(a). The first three iterations are not prefetched for, hence all references are cache misses. At each iteration from $i = 6$ onward, the read from `C[i]` is going to be a cache miss, which on our hypothetical architecture takes 50 clock cycles. This is 49 cycles more than in the original estimate, and thus $\mathit{IterTime} = 20 + 49 = 69$. Using Equation (4.1), we need $\mathit{PrefDistance} = \lceil 50/69 \rceil = 1$ iteration in advance. However, GCC schedules prefetches using $\mathit{PrefDistance} = 3$ iterations in advance, according to the original calculation. Moreover, because of this inconsistency, no prefetch instruction is inserted for the `C[i]` reference, causing a miss at every iteration as illustrated in Fig.4.3(a).

Let us examine an alternative scheduling algorithm in which a smaller *PrefetchDistance* is used. The RAP algorithm discussed in the rest of this chapter is based on this scheme.

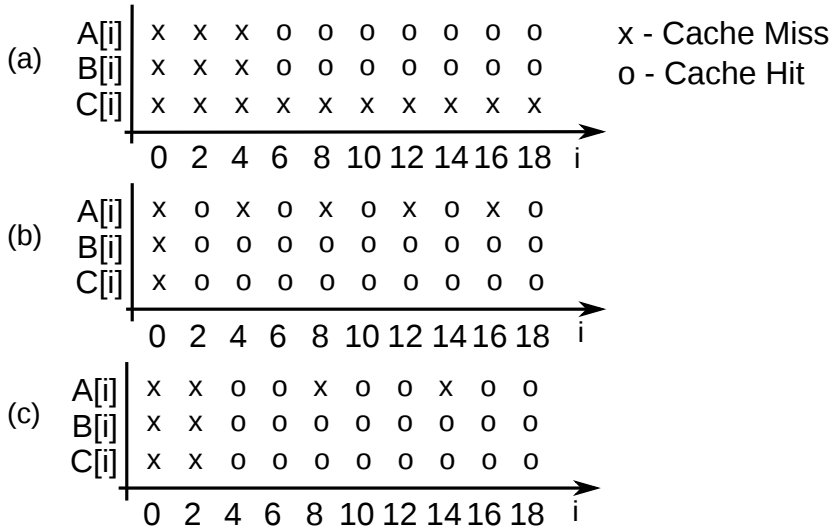


Figure 4.3: Dynamic cache trace for the code in Figure 4.2. (a) GCC loop prefetching with *PrefDistance* $= 3$ and (b) RAP algorithm with *PrefDistance* $= 1$ (c) RAP algorithm with *PrefDistance* $= 2$.

If we use *PrefDistance* $= 1$ iteration instead of 3, we can now issue prefetches for all three references, using a total of *MaxRequests* $= 3 \times 1 = 3$ prefetch slots. The cache trace for this case is shown in Figure 4.3(b). When $i = 0$, we issue prefetch requests for `A[2]`, `B[2]` and `C[2]`, then we encounter three cache misses for `A[0]`, `B[0]` and `C[0]`. For $i = 2$, we start by issuing prefetches for iteration $i + 2 = 4$, then all references are cache hits, because the prefetch requests issued at the beginning of iteration $i = 0$ overlapped with the previous misses and have had time to complete (see Figure 4.3(b)). For $i = 4$, we have a cache miss for `A[4]`, but that gives enough time for the prefetches for `B[4]` and `C[4]` to complete, and thus they become cache hits. The cache miss for `A[4]` also gave enough time for all prefetches for iteration $i = 6$ to complete, meaning we have three cache hits in that iteration. The execution enters a *steady state* at this point, with one cache miss every other iteration, until the end of the loop.

Similarly, we can also use *PrefDistance* $= 2$, which yields the code in Fig. 4.2(e) and the trace in Fig. 4.3(c). Following a similar reasoning, we observe that in the steady state we encounter one miss every 3 iterations, leading to:

Claim 1 *Let* $PD_{Mowry} = \left\lceil \frac{MissLatency}{IterationTime} \right\rceil$ *the prefetch distance computed by Mowry's algorithm (and also GCC). For any prefetch distance* $PD_{RAP} < PD_{Mowry}$ *and*

$$PD_{RAP} \times NumRefs \leq PrefetchSlots \tag{4.3}$$

we can issue prefetch instructions PD_{RAP} iterations in advance for all references without exceeding the available PrefetchSlots (number of MSHR entries), and this will result in exactly one cache miss per $PD_{RAP} + 1$ iterations in the steady state.

The claim can be easily verified: once a cache miss has been encountered, it allows enough time for all the prefetch requests already issued for the next PD_{RAP} iterations to complete, ensuring they are all hits. However, since PD_{RAP} iterations with all hits do not provide enough time to hide the miss latency, iteration $PD_{RAP}+1$ encounters a cache miss for the first read. The cycle then repeats.

Using Claim 1, we can compute the average loop iteration time in the steady state when $PD_{RAP} < PD_{Mowry}$:

$$AvgIterTime = IterHit + \frac{IterMiss - IterHit}{PD_{RAP} + 1} \tag{4.4}$$

where *IterHit* is the iteration time when all references are hits (20 cycles in our example) and *IterMiss* is the iteration time with one cache miss (69 for our example).

The average iteration time (4.4) is a strictly decreasing function of the prefetch distance PD_{RAP}. To minimize the overall execution time, we use the upper bound value:

$$PD_{RAP} = \left\lfloor \frac{PrefetchSlots}{NumRefs} \right\rfloor \tag{4.5}$$

given by (4.3). In the example in Figure 4.2(e), we have $PD_{RAP} = \lfloor 6/3 \rfloor = 2$. We can now present our improved compiler algorithm:

Case III.1 corresponds to the non-resource restricted situation, where we fall back on the same scheduling algorithm as Mowry's (and GCC) algorithm. Case III.2 occurs in situations when there are not enough *PrefetchSlots* to completely hide all cache misses; the algorithm issues one prefetch for each reference using a smaller prefetch distance, resulting in one cache miss every PD_{RAP+1} iterations. Case III.3 occurs in severely resource-constrained cases, where we have more static references than *PrefetchSlots*. The algorithm issues prefetch instructions one iteration ahead to as many references as possible, without exceeding *PrefetchSlots*.

Algorithm 4.2 Resource-Aware Prefetching

I-II. *Identical to Steps* ***I-II*** *in Algorithm 1.*

III. *Compute* $PD_{Mowry} = \left\lceil \frac{MissLatency}{IterationTime} \right\rceil$ *and NumRef the number of references. Let* $PD_{RAP} = \left\lfloor \frac{PrefetchSlots}{NumRefs} \right\rfloor$.

- ***III.1*** *If* $PD_{Mowry} \times NumRefs \leq PrefetchSlots$*, schedule prefetch instructions for all NumRef references* PD_{Mowry} *iterations in advance.*
- ***III.2*** *If* $PD_{Mowry} \times NumRefs > PrefetchSlots$ *and* $PD_{RAP} \geq 1$*, schedule prefetch instructions for all NumRef references* PD_{RAP} *iterations in advance.*
- ***III.3*** *If* $PD_{Mowry} \times NumRefs > PrefetchSlots$ *and* $PD_{RAP} = 0$*, schedule prefetch instructions for the first PrefetchSlots references in program order exactly* ***one*** *iteration in advance.*

4.4 Additional Optimizations

During the design and evaluation of the RAP compiler algorithm, we implemented a few other compiler transformations, which contributed to the performance benefits reported in our experimental results.

Thread clustering. Loop prefetching does not naturally apply to all types of workloads and data structures. However, given the nature of fine-grained parallel code – short work units, high degree of parallelism – prefetching can be enabled for some benchmarks by using the thread clustering transformation discussed in Section 3.1.6.1: the compiler inserts several short independent work units (or tasks) in a loop within a coarser task, effectively enabling the use of loop prefetching, at the possible cost of a less load-balanced execution. Thread clustering allowed us to evaluate the loop prefetching algorithm on all our benchmarks.

Prefetching after reference. In most modern architectures, a MSHR entry is allocated for a prefetch request as soon as it is issued, affecting the value of the *MaxRequest* value. Consider a memory reference `A[i]` and its associated prefetch instruction `prefetch(A[i+PD])`, with PD the prefetch distance. Right before `A[i]` is accessed we will have pending (unconsumed) prefetches for references `A[i], A[i+1],...A[i+PD-1]`, using up a total of PD MSHR entries. At this point we can chose to issue the prefetch for `A[i+PD]` before or after the reference to `A[i]`. If the prefetch is inserted textually in the

code before the reference to `A[i]` (i.e. as in Fig. 4.2), an additional MSHR will be needed (for a total of $PD + 1$). Alternatively, the prefetch instruction for `A[i+PD]` can be inserted in the code right *after* the reference to `A[i]`. Hence one prefetched value will be consumed before issuing the new one, thus require reserving only PD MSHR entries. This leads to a saving of one MSHR entry per reference prefetched, at the cost of hiding slightly less latency, since the prefetch is now issued closer to the use. In severely resource-constrained environment, this minor optimization can have a non-trivial effect on performance.

We implemented the "prefetch-after reference" optimization in the RAP algorithm, but we left Mowry's and the GCC implementations unchanged to use the default "prefetch-before" variant.

4.5 Evaluation of Prefetching Algorithm

4.5.1 Compiler and Execution Infrastructure

The XMTC compiler is based on the GNU C compiler (GCC) 4.0.2, modified to support the XMTC language extensions and to generate the code for the XMT architecture. At the highest level, the compiler consists of the following three consecutive passes:

- the prepass performs source-to-source (XMTC-to-XMTC) transformations and is based on CIL [NMRW02]. The most important operations done in the prepass are outlining spawn blocks (to prevent GCC from moving instructions across a serial-parallel switch) and identifying candidates for value broadcasting
- the core-pass performs the bulk of the compilation and is based on GCC v4.0, and
- the post-pass, built using SableCC [GH98] takes the assembly produced by the core-pass, verifies that it complies with XMT semantics and links it with external data inputs.

We implemented the **resource-aware loop prefetching algorithm** (RAP) as an optimization pass within the GCC compiler. It operates using the TreeSSA framework [Nov03] which was introduced starting with GCC 4.x releases.

The RAP algorithm operates on the code executed by each TCU while in parallel mode. It optimizes only the inner-most loops of the code, leaving outer loops or non-loop code unchanged. The pass inserts prefetch instructions that bring data from lower levels of the

memory hierarchy (shared cache or DRAM) into the prefetch buffers located at each TCU. To implement the RAP algorithm, we used some of the analyses that are part of GCC:

- *Loop induction variable analysis:* Identify loop induction variables in the code. Also used to express the address of a memory reference as an affine function in the loop induction variable if possible, identifying the base and stride.
- *Estimate the duration of a loop iteration:* Using "weights" for each type of instruction to estimate the number of cycles needed for a loop iteration. This is needed to compute the prefetch distance.

Two important parameters of the RAP algorithm are the size of the MSHR file and the latency to the shared cache. These can be read from a architecture-specific configuration file provided with the compiler, or can be specified by the user by passing specific command line arguments to the compilation process.

The RAP pass can be enabled or disabled by the programmer using command-line arguments. We have not observed any slowdowns caused by enabling the prefetching pass, but, as with any optimization pass, we recommend to the performance programmer to experiment with and without this optimization to chose the best performing configuration during the testing and optimization phases of development.

To collect running times, we used XMTSim, the XMT cycle-accurate simulator. The simulated configuration consists of 64 cores (TCUs) grouped in 8 clusters, with 256KB of shared on-chip cache and 4 DDR2 DRAM channels. The TCU MSHR file size varies between 1 and 12 entries.

Details about the design and functionality of the XMT compiler and the XMTSim simulator are presented in [KTC+11]. The source code for the RAP loop prefetching optimization is included in the XMT software release [XMT10].

4.5.2 Benchmarks

Table 4.1 describes the benchmarks used for evaluating the compiler algorithm performance. The benchmarks are written in XMTC and were chosen from a variety of domains to reflect various access patterns and application types. Our goal in collecting the benchmarks was to sample as many application domains as possible, and as a result we

Table 4.1: Benchmarks used

Name	Description	Input	*MR* [a]
jacobi	2D PDE solver kernel	1024x1024	12
lu	LU factorization	256x256	12
conv	Image convolution	128x128	12
separ	Separable image filtering	512x256	8
dbscan	SQL Non-indexed Select query	2M records	6
matmult	Dense matrix multiplication	256x256	12
SpMV	Sparse matrix - vector mult.	4M values	9
treeadd	Summation of binary tree nodes	1M values	6

[a]Maximum number of simultaneous prefetch requests required when using loop prefetching

have both integer and floating point kernels from scientific computing, image processing, databases and linear algebra.

4.5.3 Evaluation of Compiler Algorithm

To determine the effectiveness of the RAP algorithm we set out to execute our benchmarks on a series of configurations. For each benchmark, we computed the performance improvement of the RAP algorithm versus both Mowry's original algorithm and the GCC implementation when using configurations with 1, 2, 4, 6, 8, 10 and 12 MSHR file capacity.

The improvements of RAP over Mowry's algorithm and the GCC implementation are shown in Figs. 4.4(a) and 4.4(b). As we see from the two figures, the average run-time improvement across all benchmarks from our compiler algorithm ranges from 25.63% to 40.15% when compared to Mowry's algorithm, and from 13.18% to 34.79% when compared to GCC, depending on the number of MSHR entries.

Fig. 4.4(b) shows the comparison of the RAP algorithm with the GCC implementation of loop prefetching. For each configuration, we provide GCC with the exact size of the MSHR file as the number of *PrefetchSlots*. When not enough *PrefetchSlots* are available to hide all latencies, the GCC algorithm does not issue prefetch instructions for some of the references. By contrast, the RAP algorithm decreases the prefetch distance, and issues as many prefetch instructions as the MSHR has capacity. This allows it to hide more of the memory latencies, and to outperform GCC.

The reason that the run-time improvements only show up to 12 words MSHR capacity can be identified by looking at the parameters of the architecture and benchmarks. On XMT, we are prefetching from the shared L1 cache to the TCUs. The latency for an L1

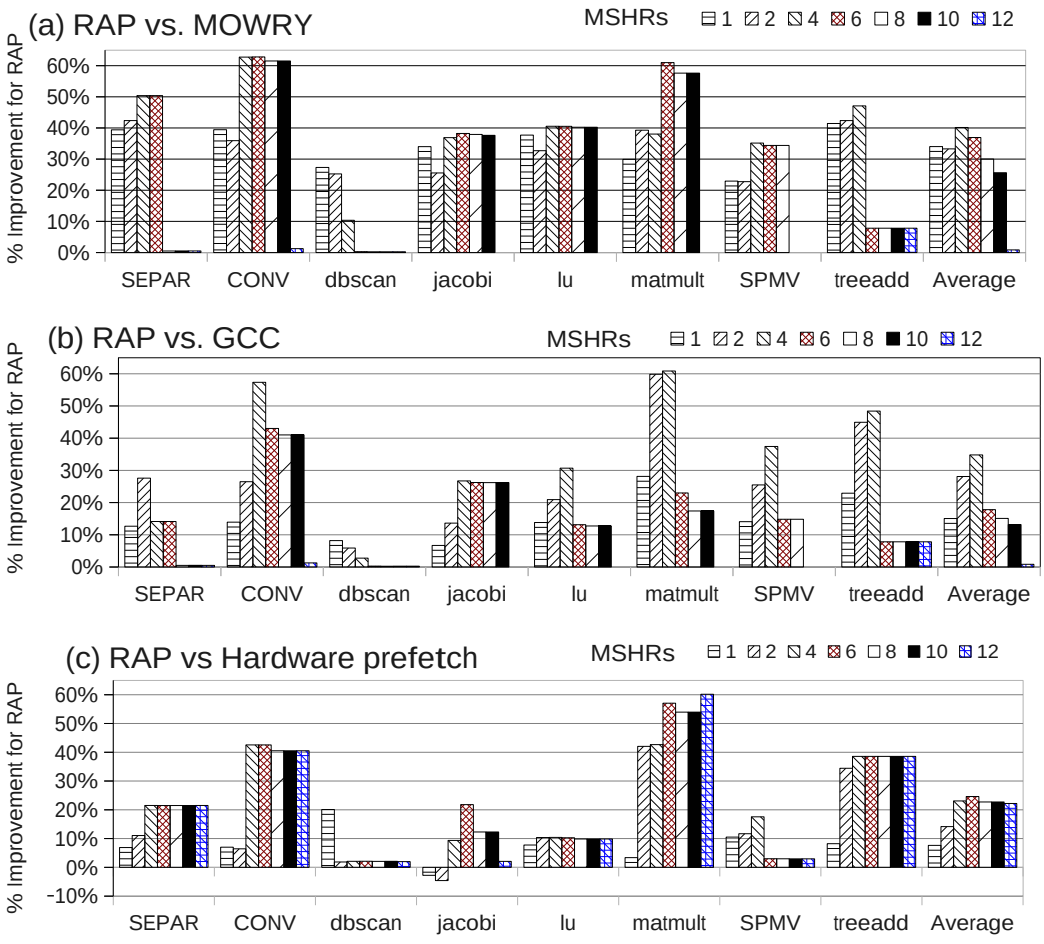


Figure 4.4: Performance improvement of RAP compared to (a) Mowry, (b) GCC and (c) one-block-lookahead hardware prefetching

access is ≈ 24 cycle in the current configuration. Given this latency, the prefetch distance is usually small (1-3 iterations), and thus the *MaxRequest* value for the benchmarks ranges between 6 and 12 for our benchmarks, as shown in Table 4.1. Therefore, for MSHR files with 12 entries or larger, we are not in a resource-constrained regime, and there are no advantages for using the RAP algorithm over the alternative algorithms.

Our results mark a significant improvement over existing approaches, as most of the time will be spent in resource-constrained conditions. For the class of highly parallel architectures we are targeting, a per-core MSHR of 12 entries represents a significant budget of area and power. Future manycore architectures will probably devote even fewer resources for the MHA, making the RAP algorithm highly relevant.

4.5.4 Comparison with Hardware Prefetching

We compare the RAP software prefetching algorithm with an implementation of XMT that includes a hardware prefetching mechanism. Traditional single- and multi-core processors include sophisticated hardware prefetching units, capable of monitoring and distinguishing multiple independent streams of requests and identifying large access strides. However, the hardware complexities of such units make them prohibitively expensive per-core for a many-core architecture. Only a simple hardware prefetcher, that requires minimal hardware additions, could be considered. A well known such technique is One-Block-Lookahead (OBL,[Smi82]), also called sequential prefetching [DDS95], which prefetches the *next cache line* once a particular line is first read.

We implemented this scheme in XMTSim, the XMT cycle-accurate simulator. Since TCUs have no regular caches (to avoid coherence costs and area constraints), we prefetch at the granularity of one word, instead of one cache line: once a read request for address x is issued, a prefetch request for address $x + 4$ is automatically generated.

To evaluate the relative performance of the hardware scheme, we execute the same set of benchmarks, introduced in Section 4.5.2. In the first experiment, we compared the software-only RAP scheme to the hardware-only OBL approach. The results in Figure 4.4(c) show that the software RAP prefetching algorithm outperforms the OBL hardware scheme by 7.64% to 24.61% on average.

In Figure 4.5 we compared the performance of the software-only RAP algorithm with configurations in which both hardware and software prefetching were enabled. Most mod-

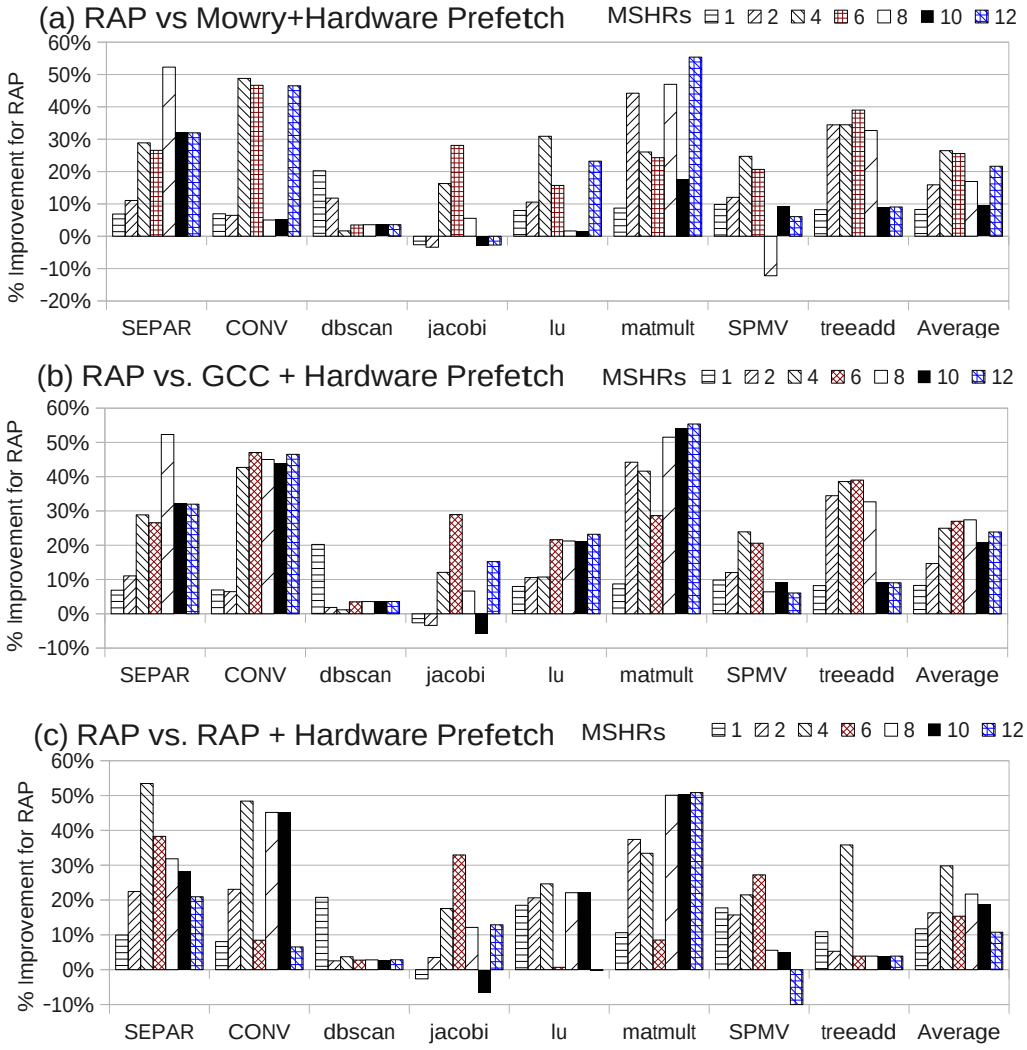


Figure 4.5: Interference of software with hardware prefetching. Performance improvement of RAP compared to (a) Mowry + OBL hardware prefetching, (b) GCC + OBL hardware prefetching and (c) RAP + OBL hardware prefetching.

ern serial and multi-core architectures have support for both, and they can usually be enabled or disabled through system configuration tools and compiler flags.

Figures 4.5(a) and 4.5(b) present the performance improvements of RAP versus configurations where on top of the OBL hardware prefetcher (OBL-HP), the compiler used Mowry's and GCC's software prefetching algorithm respectively. The hardware and software prefetchers interfere and compete for using the MSHR, which leads to less predictable results. In particular, on few configurations, the combined software-hardware outperforms the software-only approach. However, on average across all benchmarks, the RAP algorithm is 8.3-26.5% faster than Mowry plus OBL-HP and 8.3-27% faster than GCC plus OBL-HP.

We also examined the performance of the RAP algorithm when the OBL-HP was enabled, as shown in Fig. 4.5(c). We observed the same performance degradation caused by the interference of the two mechanisms, with the software-only RAP outperformed RAP plus OBL-HP by 10.8-29.8%.

This strengthens the case that given the severe per-core limitations present in many-cores, least resource-intensive latency hiding techniques such as software prefetching offer the best performance.

4.6 Sensitivity Analysis of the RAP Algorithm

In the previous sections of this chapter, we described a compiler prefetching algorithm that adapts to the available hardware resources and is optimized for the same. The compiler is part of the link between the Programming and Execution models discussed in Chapter 3. This section focuses on the Execution model, and provides a framework for selecting the optimal parameters for the execution model, as a tool to the hardware designers. Even though we restricted the study to only two architectural parameters, the framework can be used for other components as well.

This section presents a design-space exploration (DSE) of the XMT architecture with respect to the two components that are most relevant to memory-level parallelism:

- the capacity of the TCU MSHR file, and
- the off-chip bandwidth to DRAM, as controlled by the number of DRAM channels.

More MSHR entries allow more data elements to be simultaneously prefetched, and more loop iterations before they are actually needed, hiding more of the latency. More DRAM channels provide more bandwidth which can be used to prefetch data, in addition to serving the regular, non-prefetched memory accesses.

The goals of this study are:

- getting a rough estimate of what the overall hardware cost for software-prefetching support is for a lightweight manycore, and whether this cost is feasible;
- understanding the trade-offs between two different hardware-assisted techniques for boosting MLP – software-prefetching and multiple DRAM channels – and determine what combination is a good configuration for a lightweight manycore architecture such as our experimental platform; and
- evaluating the ability of the RAP software prefetching algorithm discussed in this chapter to achieve good performance for a wide variety of machine configurations in an adaptable manner.

4.6.1 Design Space

We consider multiple design space points that range from the bare-bones architecture with no MSHR file and one DRAM channel, to a configuration including 20-word MSHR files and 8 DRAM channels. We scale the capacity of the MSHR file linearly in increments of 2 words per core, and use powers of two for the number of DRAM channels (evaluating 1, 2, 4 and 8).

We did not include the hardware required to implement the One-Block-Lookahead hardware prefetching mechanism in the design space. This scheme requires additional per-TCU hardware to compute next block's address and issue the new prefetch request. A MSHR file is still required to keep track of the outstanding memory requests issued but not completed. The results presented in Section 4.5.4 show that the OBL scheme is always slower than the RAP software approach while requiring more chip resources, therefore there is no benefit in choosing a configuration with OBL support.

4.6.2 Targeted ASIC Parameters

Our basic design is based on the HDL description of the XMT Paraleap system [WV08a]. The HDL design was validated by prototyping using both FGPA and ASIC technology. The DSE results in this section are based on the ASIC synthesis process, for which we used the Synopsys Design Compiler and the 90nm IBM CMOS9SF library, part of Artisan's SAGE-X v3.0 Standard Cell Library.

The targeted clock frequency of a 64-TCU XMT ASIC is in the same range as a modern many-core architecture such as GPUs. This claim is based on the results from (i) the ASIC implementation of the Mesh-of-Trees (MoT) interconnection network fabricated in 90nm IBM technology [BHQV07], (ii) a complete 64-TCU XMT integer-only chip using the same IBM fabrication process and (iii) a detailed chip area comparison between a 1024-XMT configuration and a NVIDIA GTX 280 [CKTV10]. By using the same generation technology as a modern GPU, as well as a comparable amount of engineering and optimization effort, XMT can support clock frequencies in the range of 1-1.3GHz and possibly higher.

4.6.3 Area Scaling Methodology

The design is highly parametrized, and therefore we can synthesize gate-level descriptions of various configurations. After synthesis, we extracted the cell area of units of interest and reference it relative to the total design cell count. A cell is the basic unit of ASIC design, a standard arrangement of transistors that implements a gate or flip-flop. We note that these area numbers are likely to change when the design undergoes the Place-and-Route process. At the present time, our tools did not permit us to obtain direct area numbers for sub-components after the P&R phase, and thus we use the cell areas reported by the synthesis as approximations of the final gate counts.

To evaluate the change in area for each of our design points, we first identify the hardware modules we are going to scale in the HDL description, determine the area they use in a representative subset of configurations from the synthesis output, then extrapolate to the the rest of the points.

4.6.4 Benchmarks

For the experiments in this chapter, we used the same set of benchmarks introduced in Section 4.5.2. These benchmarks cover a broad area of application domains and types of workloads, reflecting our goal to evaluate and optimize XMT as a general-purpose platform.

4.6.5 Results

We evaluate the chosen design points from the point of view of average runtime across all the benchmarks. For each hardware design configuration described in Section 4.6.1, we initialize the XMTSim cycle-accurate simulator with the respective set of parameters and execute all the benchmarks in the test suite. We use the RAP compiler algorithm described in Chapter 4, as well as the thread clustering transformation discussed in Section 3.1.6.1 to enable prefetching for all the benchmarks in our study.

As reference point, we use a bare-bones configuration that includes 64 parallel TCUs, no MSHR file and one DRAM channel. For each design point, we evaluate the relative increase in area due to the change in parameters (MSHR file capacity or additional DRAM channels). Figure 4.6 shows the observed increase in number of cells used for the DRAM channels and for the MSHR File when scaling the number and capacity respectively. For example, we identified that the area used for increasing the capacity of the MSHR file from 4 to 8 words over all 64 TCUs is 263,872 cells, representing 0.59% increase from the base configuration total chip area of 44,797,832 cells. An additional DRAM channel added 357,819 cells, representing a 0.8% increase in area relative to the same base configuration. Except for the non-linear increase observed between zero and four MSHR entries, due to fixed costs of adding the prefetch datapath, the growth is mostly linear. We extrapolate from our set of synthesized configurations to the rest of the design space points to estimate area increases.

We record the average execution time across all the benchmarks, normalized to the bare-bones configuration. Figure 4.7 represents a Performance-Area scatter plot of all the design points. For each point, we include a 2-tuple representing the number of DRAM channels and the MSHR file capacity. The point labeled (1,0) is the reference configuration used, normalized to a runtime of 100 and area increase of 0%. A first insight is the signifi-

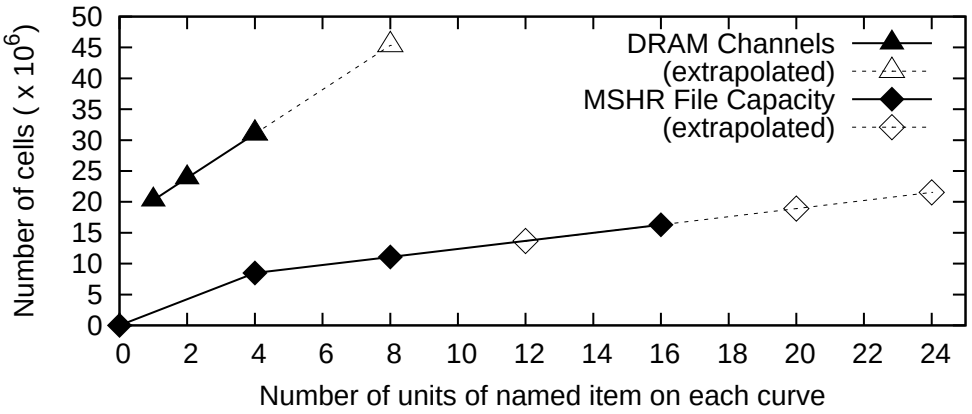


Figure 4.6: Number of area cells used for the DRAM channels and MSHR File when scaling the number and capacity respectively.

cant increase in performance resulting from the inclusion of even a small MSHR file. This is illustrated by the difference in performance between the configurations with no MSHR file (1,0), (2,0), (4,0),(8,0) and the rest.

We combined collected average performance numbers with area information and determine the *Pareto-optimal* design space points. These represent the only viable choices to the hardware designer, since no other configuration has better performance for a lower area. Note that this curve is computed using a heterogeneous set of benchmarks. For a special-purpose architecture, the same methodology can be used, but using a workload that is characteristic of the target application domain, and a different curve will be obtained.

The dotted line in Figure 4.7 depicts the *Area-Performance Pareto frontier.* A system designer can use this diagram to choose the best performing configuration for a given area, by choosing the appropriate Pareto optimal point. In addition, we can observe the knee of the curve (4,12) which is the configuration point with 4 DRAM channels and 12 MSHR file entries, and represents *the point of diminishing returns*, with 5.47% area increase and 53.66% performance improvement; after reaching point, additional increases in area do not translate in significant increases in performance. This knee of the curve (4,12) represents our best recommended configuration for an XMT implemented with today's technology.

Hardware engineers can use the data in Figure 4.7 to make informed design decisions. For example, they can determine that by increasing the total chip area by 3.09%, the resulting system will be 47.36% faster (for the 2 DRAM channels, 4 entry MSHR file configura-

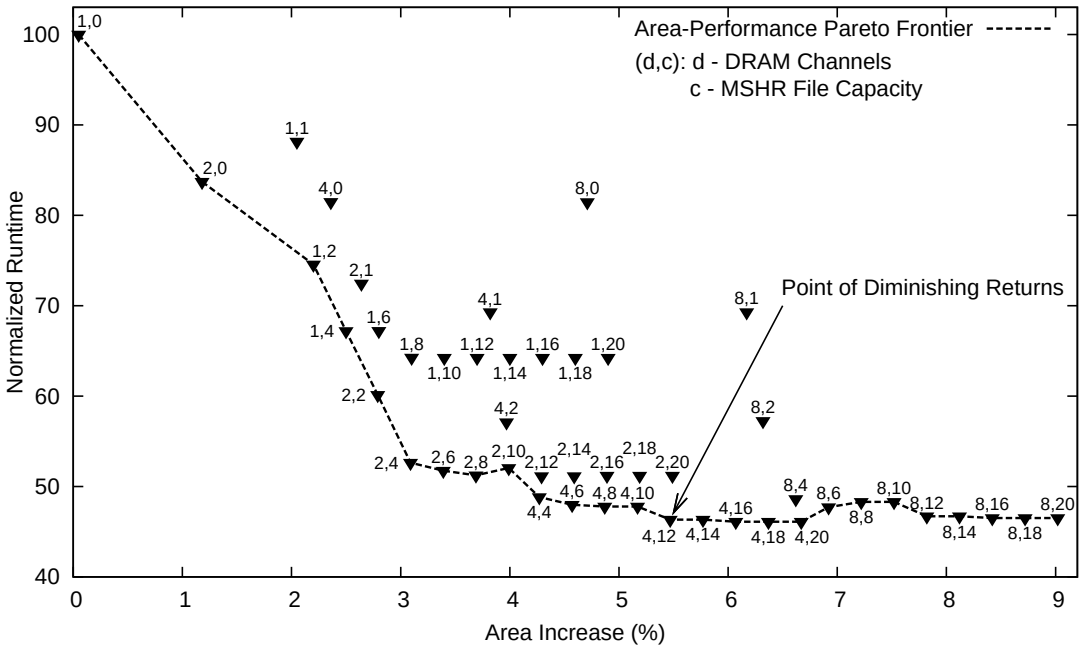


Figure 4.7: Area and performance change when varying the number of DRAM channels and the capacity of the MSHR file. Each point includes a 2-tuple (d,c) representing (No. DRAM channels, MSHR file capacity).

tion), observation that can affect the decision to allocate additional resources or use more aggressive optimizations in the design and synthesis tools.

4.6.6 Additional Considerations

We are using a number of simplifying assumptions in this study by not including considerations such as the I/O pin count and power/thermal optimizations. Such topics are currently under study as part of the XMT project and we plan to incorporate them as additional dimensions in future work. This study is still useful, however; since if, for example, there is an upper bound on the allowed pin count, the designer can choose among a subset of all the Pareto-optimal designs we derive; namely the subset that also satisfies the pin count constraint.

The results in Figure 4.6 are dependent on the benchmarks used to collect performance results. In this case, they include results from a number of application domains, with different characteristics. One interesting use of our study is optimizing the architecture for a specific application or domain, e.g. medical image processing or network packet processor. To accomplish that, a different set of benchmarks that reflect the specific type of

workload can be used. Using these benchmarks, the exact same methodology presented here can be followed to tune some of the architectural parameters of such a chip.

4.7 Related Work

Prefetching is a widely studied technique used to hide the increasingly high latencies (in terms of clock cycles) of memory accesses in modern architectures. Software prefetching [MLG92, McI98, CMCH91] relies on the existence of non-blocking prefetch instructions and is usually enabled by the compiler. In hardware prefetching (e.g. [Smi82, Jou90, LRB01, CB95, DDS95]) a specialized hardware unit infers prefetching opportunities by monitoring run-time behavior. Prefetching schemes for parallel architectures in both software [McI98, TE95, Mow98] and hardware (e.g. [DDS95]) build upon uni-processor prefetching by taking into consideration issues caused by sharing of data and resources, such as coherence traffic and overheads.

Several studies have considered the interaction of the architectural parameters with the performance of software prefetching algorithms. In his comprehensive work on software data prefetching, Mowry [Mow95] explores the effect on execution time of varying the number of outstanding prefetch requests that can be handled simultaneously by the hardware. The author also compares two versions of the prefetch issue buffer hardware - one in which the processor stalls when the buffer is full, and one where additional requests are simply dropped. Their results were mixed, with the former performing slightly better when using the prefetching algorithm described in Section 4.2. In follow-up work, Mowry [Mow98], as well as McIntosh [McI98], settle for a fixed-size prefetch issue buffer of 16 locations. Several other papers study the effects of changing the size of the prefetch destination (either cache or dedicated prefetch buffers) for systems with software [CMCH91, KL91] or hardware prefetching [Jou90]. However, unlike our approach, in all of these existing schemes the prefetch algorithm is unaware of the prefetch hardware configuration, and does not adapt its behavior.

GCC is the only attempt to consider the amount of prefetch resources available as part of the loop prefetching algorithm. As described in Section 4.2, the GCC algorithm limits the number of memory references prefetched to meet a fixed upper bound. However, no guidance is given on how to chose this upper bound, as it is not clear what the un-

derlying hardware limitation is accounted for. Yang et. al [YYX05] empirically set the maximum number of prefetch instructions issued by the GCC compiler for the IA64 platform to 12. However, their study does not address the underlying limitations of the GCC algorithm discussed above. In our approach, we identify the hardware resource dictating the maximum number of prefetch requests allowed (the MSHR file), and provide an original scheduling algorithm which limits the prefetch distance instead of the number of references prefetched, and show that it outperforms both Mowry's and GCC's implementations.

In recent work, Tuck et. al [TCT06] propose an alternate MHA organization that would provide increased MSHR capacity at the expense of slightly higher latencies. However, it is not clear how the performance of the scheme would change when scaled to the degree of parallelism required by the emerging manycore architectures. Different schemes for improving the functionality of existing MHAs, by either dynamically adjusting the MHA capacity to reflect the memory bus load [JN09] or by devising a MHA-aware cache replacement policy to reduce the number of cache misses [QLMP06] have been proposed. Our compiler algorithm is orthogonal to these techniques and can function alongside these hardware enhancements.

In the area of design-space exploration, recent years have seen a burst of studies targeted at understanding the interactions between performance, energy, thermal efficiency and area in the design of chip multi-processors (CMPs), or multi-cores. Huh et. al [HBK01] explore the design space of CMPs as the available transistor budget grows with the fabrication technology. They consider in-order vs. out-of-order cores, amount of cache per processor and availability of off-chip bandwidth, while varying the area constraints. Li et. al [LLB+06] extend the study by including pipeline depth and width, operating voltage and frequency and cooling mechanisms as design space dimensions, as well as adding thermal constraints. Methods for automatically or semi-automatically design space exploration for specialized architectures such as System-On-Chip and signal processing have also been proposed [GVH01, LvdWDV99]. These studies are all focused on share-nothing workloads, optimizing architectures designed for throughput rather than single-task completion time.

The research presented in this chapter differs from prior work in that it focuses on MLP resources – MSHR capacity and DRAM channels – for design space exploration. It also

does so in conjunction with a resource-aware compiler algorithm that efficiently uses the limited resources at each design point. This study is targeted at the needs of lightweight manycore architectures which aim to optimize single-task completion time, an emerging architectural paradigm.

Chapter 5

Performance on Irregular Benchmarks

In the previous chapters, we discussed a programmer's workflow and additional system support for optimizing the performance of application code written for the XMT architecture. This chapter applies and evaluates the contributions of this thesis by comparing overall performance of programs written for XMT with existing serial and parallel solutions.

The comparison consists of two studies. In the first study, discussed in Section 5.1, we evaluate the performance of the 64-core Paraleap XMT prototype to a multi-core architecture, namely an Intel Core 2 Duo. The methodology of this study, including the benchmarks, the datasets and the measure of performance used for the comparison were suggested to us by a senior individual from Intel Corporation.

The second study consists of a comparison between XMT benchmarks simulated on a 1024-core XMT configurations using XMTSim and parallel programs written using the CUDA framework and executed on NVIDIA Tesla GPU architectures. The study is discussed in Section 5.2.

In the course of this chapter we have:

- used the programmer's workflow of Chapter 3 while programming the benchmarks;
- applied the Resource-Aware Prefetching compiler algorithm introduced in Chapter 4 in the XMTC compiler; and
- configured the parameters of the XMT architecture to maximize performance with the minimum amount of chip resource, following the methodology of Section 4.6.

In this chapter we focus on the performance of relatively simple benchmarks (sometimes called kernels). These kernels are usually part of larger applications. However, optimizing kernels is extremely important, as in many cases most of the execution of a larger application is spent within a small number of such kernels. In Chapter 6 we conduct a similar experiment, but using an application that is about an order of magnitude more complex than the kernels discussed here.

5.1 Comparison with a Modern Serial Architecture

In this Section we discuss a comparison of the 64-core FPGA-based Paraleap prototype of the XMT architecture and an Intel Core 2 Duo processor.

Programming for these two platforms requires using different programming models, frameworks and tools. The XMT model guides the programmer to start by designing a parallel algorithm, then express all the available parallelism in the resulting XMTC program. The architectures is designed to use all the available cores to efficiently execute the parallel code. On the other hand, the current multi-core architectures (such as the one considered in this study) consist of a small number of essentially serial cores placed on one die, with no clear algorithmic model to be followed in designing efficient implementations. While there are some attempts to offer programming frameworks which map to these platforms (e.g. POSIX and Java threads, OpenMP [Ope08], TBB [Rei07], Cilk [FLR98]), the most common methodology followed by programmers at this time is to write a serial program, and rely on system software and hardware to take advantage of the additional cores. For this experiment, we used a similar methodology: we used optimized serial implementations, and enabled compiler optimizations such as automatic parallelization and vectorization, data prefetching and instruction reordering to take advantage of the parallelism available in hardware.

We describe our experimental setup and methodology next.

5.1.1 Benchmarks

The benchmarks used for the comparison with the Intel platform were suggested to us by the Intel engineer as constituting a representative set of workloads. The applications we used were:

- **SpMV:** A sparse matrix, stored in Compact Sparse Row (CSR) format, is multiplied by a dense vector. The implementation is straightforward: the serial version simply multiplies each row with the vector one at a time, while the parallel implementation processes all rows in parallel, using exactly one thread per row.

- **FFT:** 1-D Fast Fourier Transformation. We used the Radix-2 Cooley-Tukey [CT65] algorithm as the basis of our implementations. The algorithm runs in stages; first, a

	Small		Large	
Program	N	Footprint	N	Footprint
SpMV	22K	200KB	4M	33MB
FFT	8K	192KB	4M	96MB
Quicksort	100K	781K	20M	153MB

Table 5.1: Datasets used. Footprints represent the total amount of memory used at runtime.

"twiddle" table, used to combine the input and output of stages, is computed. Next, a binary bit reversal pass prepares the input data. The main part of the algorithm consists of $\log N$ "butterfly" computation stages. The serial implementation follows this algorithm. For the parallel version, we parallelized each of the stages using the XMT parallel programming model. Note that at the present time, Paraleap has only integer arithmetic support; to allow for a fair comparison, we implemented both the serial and parallel versions using fixed-point arithmetic with the same precision.

- **Quicksort:** Sorting an array of integer values. For the serial implementation of this benchmark, we implemented the standard Quicksort algorithm found in any serial algorithms textbook. The parallel version of quicksort follows the algorithm introduced in [HNR90]: in the first phase, the array is iteratively partitioned using a fetch-and-add based parallel scheme, taking advantage of the XMT prefix-sum primitive. When the number of partitions exceeds a threshold, the execution switches to its second phase, where each partition is sorted by exactly one thread.

As a result of the coding effort described above, we had two implementations for each benchmarks:

1. a **parallel** program, written in XMTC to be executed on Paraleap and
2. a **serial** C program for the Intel Core 2 Duo architecture.

Table 5.1 describes the **input data** used in our experiments. For each benchmark, we created two datasets: a small one, that is comparable in size to the on-chip cache, and a larger one that exceeds the cache size for both Paraleap and the Intel Core 2 Duo processor.

5.1.2 Experimental Setup

We collected cycle counts for the parallel XMTC programs running on Paraleap and for the serial C implementations on the Intel Core 2 Duo system as follows:

Parallel execution We compiled each benchmark using the XMTC compiler, which is a port of GCC 4.0.2 to the XMT platform. We used the maximum level of optimization supported (-O3), and activated data prefetching optimizations tuned for XMT. In addition, we manually added instructions to use the cluster read-only buffers of the XMT architecture. We expect this optimization to be automated in a future version of the compiler by performing analyses to identify data that can be cached in these buffers. We ran the compiled benchmarks on the 64-core Paraleap FPGA prototype and collected the cycle counts reported by the system.

Serial execution For the second part of our experiment, we used a desktop system with an Intel Core 2 Duo E6300 CPU rated at 1.86GHz, with 64KB L1 and 2MB L2 cache per processor and 2GB DDR2-667 DRAM. Choosing a compiler for the Intel x86 architecture can have a significant influence on execution performance, since different compilers utilize the large array of features of the architecture in different ways. We used two compilers in our experiments: (i) the widely used open-source GNU C Compiler (GCC) and (ii) the Intel C++ Professional Compiler for Linux (ICC v11.0), the most recent version at the time of writing. Just as for the Paraleap compiler, we used the highest level of optimization for both compilers. In addition, we enabled the advanced optimizations available for the ICC compiler on the Core 2 Duo architecture: SIMD vectorization using SSE3, software data prefetching and auto-parallelization. We ran each benchmark 5 times and collected cycle counts using the *Time Stamp Counter* 64-bit register instead of the more coarse-grained system timers. This ensured better precision and factored out OS interferences.

An FPGA prototyping/cycle-accurate emulation methodology for projecting the cycle counts for an 800MHz ASIC implementation, using DDR2-800 SDRAM was introduced in [WV08b]. We are using the same configuration for our current experiments.

5.1.3 Results

We collected the cycle counts for the XMT parallel implementation, as well as cycle counts for the serial implementation using both the GCC and the ICC compiler. For each benchmark, we computed four speed-up results:

icc-small: On the *small* dataset, using the ICC compiler for the serial execution

gcc-small: On the *small* dataset, using the GCC compiler for the serial execution

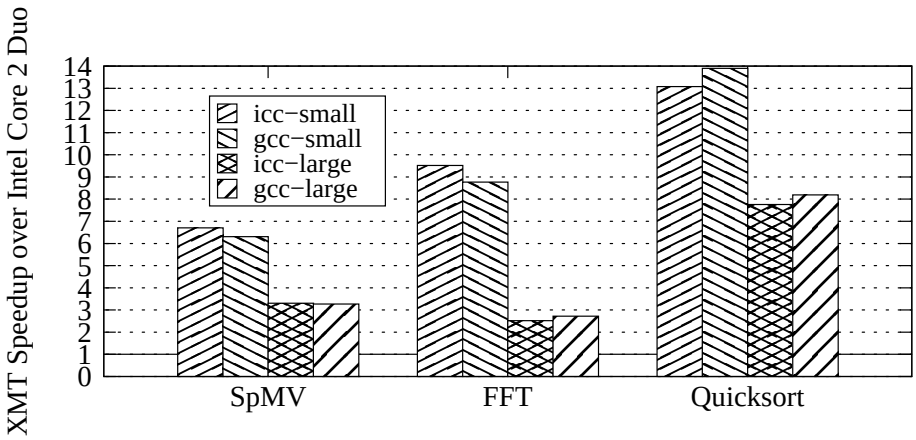


Figure 5.1: Speedups of the 64-TCU Paraleap XMT prototype vs. Intel Core 2 Duo

icc-large On the *large* dataset, using the ICC compiler for the serial execution

gcc-large On the *large* dataset, using the GCC compiler for the serial execution

The speed-up figures represent the ratio between the **number of clock cycles** needed for the execution of each of the benchmarks on the Paraleap and the Intel Core 2 Duo computers. The speed-ups for the three benchmarks are presented in Figure 5.1.

The lower speed-up numbers for the *large* datasets can be explained by the large difference in cache size between Paraleap and Core 2 Duo (256KB compared to 2x2MB). However, there is no reason that the cache size of a future XMT system will be smaller than its contemporaries, addressing this discrepancy.

In terms of silicon area, the comparison above is tilted in favor of the Intel design. The silicon area of an ASIC implementation of the 64-processor XMT design is roughly the same as that of a *single* core of the Intel Core 2 Duo. The same holds with regards to the compiler, since both compilers used for the Intel platform are mature, established products, while the XMTC compiler performs only basic optimizations at this stage.

Overall, we observed speedups in the ranging between 6.3x and 13.89x for the datasets that fit in the cache, and 2.5x to 8.18x for datasets exceeding the size of the cache on both platforms.

These speedups are reported in terms of cycle counts. An important reason for the comparison is getting a feel for the scalability potential of XMT relative to possible up-

grades of the Intel. We expect the speedups to improve significantly with the increase in the number of XMT cores, for example when using a 1024-core configuration instead of the 64-core considered. These are encouraging results, as we do not see why the clock speed of a 1024-TCU XMT should be lower than a same-generation many-core processor that incorporates cache-coherence and uses the same silicon area. A parallel effort part of the XMT group makes this case for larger configurations [CKTV10].

5.2 Comparison with a Graphics Processing Unit (GPU)

In this section, we present a meaningful performance comparison of a state-of-the-art GPU to XMT, on a range of irregular applications. We show via simulation that XMT can outperform GPUs on these applications, while not falling behind significantly on regular ones.

5.2.1 Tesla Framework

In the recent years, the GPU architectures have evolved from purely fixed-function devices to increasingly flexible, massively parallel programmable processors. The CUDA programming environment (e.g. [NBGS08, NVI09]) together with the NVIDIA Tesla [LNOM08] architecture is one example of a GPGPU system gaining acceptance in the parallel computing community. In commercial products, the names G80 and GT200 refer to the two existing implementations of the Tesla architecture.

Fig. 5.2 depicts an overview of the Tesla architecture. It consists of an array of Streaming Multiprocessors (SMs), connected through an interconnection network to a number of memory controllers and off-chip DRAM modules. Each SM contains a shared register file, shared memory, constant and instruction caches, special function units and several Streaming Processors (SPs) with integer and floating point ALU pipelines. SFUs are 4-wide vector units that can handle complex floating point operations. The CUDA programming and execution model are discussed elsewhere [LNOM08].

A CUDA program consists of serial parts running on the CPU, which call parallel *kernels* offloaded to a GPU. A kernel is organized as a grid (1, 2 or 3-dimensional) of thread blocks. A *thread block* is a set of concurrent threads that can cooperate among themselves through a block-private shared memory and barrier synchronization. A global scheduler assigns thread blocks to SMs as they become available. At each instruction issue time, a

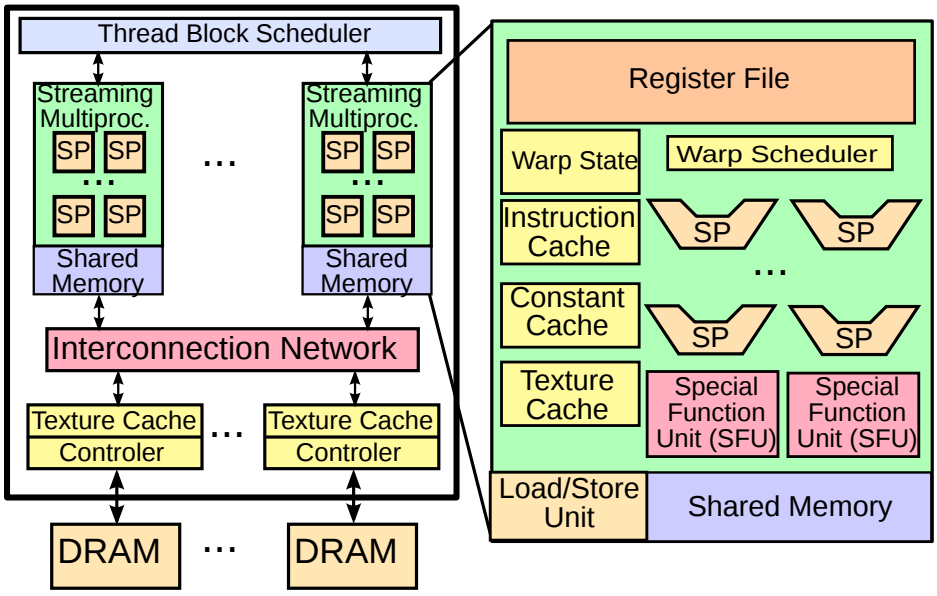


Figure 5.2: Overview of the Tesla Architecture

scheduler selects a fixed-size warp (32 threads) that is ready to execute and issues the next instruction to all the threads in the warp. Threads proceed in lock-step manner, and this execution model is called SIMT – Single Instruction Multiple Threads.

The CUDA framework provides a relatively familiar environment for developers, which led an impressive number of applications to be ported since its introduction [NVI10]. Nevertheless, a non-trivial development effort is required when optimizing an application in the CUDA model. Some of the considerations that must be addressed in order to get real performance gains follow.

- *Degree of parallelism:* a minimum of 5,000 - 10,000 of threads need to be in-flight for achieving good hardware utilization and latency hiding.

- *Thread divergence:* in the CUDA Single Instruction Multiple Threads (SIMT) model, divergent control flow between threads causes serialization, and programmers are encouraged to minimize it.

- *Shared memory:* no standard cache is included at the SM in the Tesla architecture. Instead, a small user-controlled scratch-pad shared memory is provided. Note that we do not classify the constant and texture caches as regular caches since they are read-only and use separate address spaces. This changed in the NVIDIA Fermi architecture, which includes an option to use part of the shared memory as an L1

cache. At the time of this writing, it is not clear yet what implications this has on programmability and performance. What limited our choice of a GPU architecture for the comparison in this work was the availability of third party application code.

- *Memory request coalescing:* better bandwidth utilization is achieved when data layout and memory requests follow a number of temporal and spatial locality guidelines.
- *Bank conflicts:* concurrent requests to one bank of the shared memory incur serialization, and should be avoided in the code, if possible.

5.2.2 Comparison of Architectures

The key issues that affect the design of both architectures, and the main differences between them, are summarized in Table 5.2.

	Tesla	XMT
Memory Latency Hiding and Reduction	·Heavy multithreading (requires large register files and state aware scheduler) ·Limited local shared scratchpad memory ·No coherent private caches at SM or SP	·Large globally shared cache ·No coherent private TCU or cluster caches ·Software prefetching
Memory and Cache Bandwidth	·Memory access patterns need to be coordinated by the user for efficiency (request coalescing) ·Scratchpad memories prone to bank conflicts	·Relaxed need for user-coordinated DRAM access due to caches ·Address hashing for avoiding memory module hotspots ·High bandwidth mesh-of-trees interconnect between clusters and caches
Functional Unit(FU) Allocation	·Dedicated FUs for SPs and SFUs ·Less arbitration logic required ·Higher theoretical peak performance	·Heavy FUs (FPU and MDU) are shared through arbitrators ·Lightweight FUs (ALU and branch unit) are allocated per TCU. ALUs do not include multiply/divide functionality
Control Flow and Synchronization	·Single instruction cache and issue per SM for saving resources. Warps execute in lock-step (penalizes diverging branches) ·Efficient local synchronization and communication within blocks. Global communication is expensive ·Switching between serial and parallel modes (i.e. passing control from CPU to GPU) requires off-chip communication	·One instruction cache and program counter per TCU enables independent progress of threads ·Coordination of threads can be performed via constant time prefix-sum. Other forms of thread communication are done over the shared cache ·Dynamic hardware support for fast switch between serial and parallel modes and load balance of virtual threads

Table 5.2: Implementation differences between XMT and Tesla. FPU and MDU stand for floating-point and multiply/divide units respectively.

5.2.3 Benchmarks and Datasets

A "general-purpose" architecture should provide good performance on both regular and irregular applications. This guided the selection of benchmarks for this study, as listed in Table 5.3. We selected benchmarks whose GPU results are published and CUDA source code made available by authors. This ensures that we are using the most optimized code for the CUDA implementation, highly tuned for GPUs. The XMT implementations of the benchmarks were developed by members of the XMT project.

Note that all our benchmarks use single-precision floating point arithmetic only, to allow for a fair comparison with Tesla. The addition of better support for double precision in upcoming GPUs will not significantly change the relative results, as the same support can be added to XMT using similar additional resources.

Name	Description	CUDA implementation source	Lines of Code		Dataset	Parallel sectn.		Threads/sectn.	
			CUDA	XMT		CUDA	XMT	CUDA	XMT
Bfs	Breadth-First Search on graphs	Harish and Narayanan [HN07], Rodinia benchmark suite [CBM+09]	290	86	1M nodes, 6M edges	25	12	1M	87.4K
Bprop	Back Propagation machine learning algorithm	Rodinia benchmark suite [CBM+09]	960	522	64K nodes	2	65	1.04M	19.4K
Conv	Image convolution kernel with separable filter	NVIDIA CUDA SDK [NVI09]	283	87	1024x512	2	2	131K	512K
Msort	Merge-sort algorithm	Thrust library [HB09, SHG09]	966	283	1M keys	82	140	32K	10.7K
NW	Needleman-Wunsch sequence alignment	Rodinia benchmark suite [CBM+09]	430	129	2x2048 sequences	255	4192	1.1K	1.1K
Reduct	Parallel reduction (sum)	NVIDIA CUDA SDK [NVI09]	481	59	16M elts.	3	3	5.5K	44K
Spmv	Sparse matrix - vector multiplication.	Bell and Garland [BG09]	91	34	36Kx36K, 4M non-zero	1	1	30.7K	36K

Table 5.3: Benchmark properties

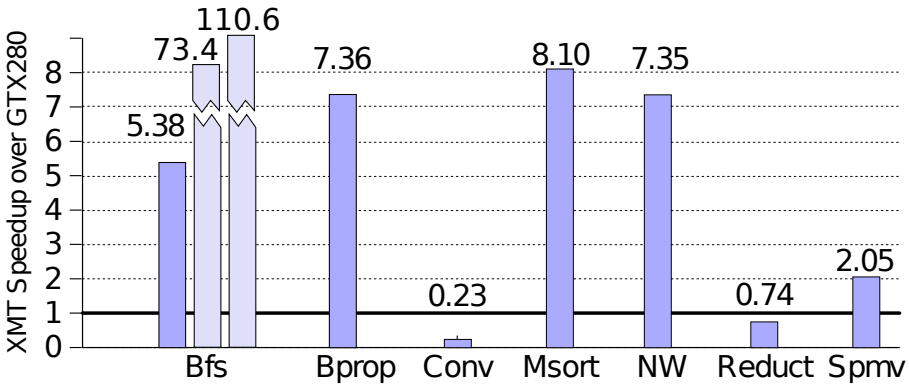


Figure 5.3: Speedups of the 1024-TCU XMT configuration with respect to GTX280. A value less than 1 denotes slowdown.

5.2.4 Tested Configurations

The premise of this study is to perform a *meaningful* performance comparison: assume the same amount of chip resources (area and power), similar fabrication technology and comparable amount of optimization. More concretely, we needed to determine the power-of-two configuration of XMT whose chip resources are in the same ballpark as the GTX 280, the GPU considered. We based our estimation on the detailed data from the ASIC implementation of the MOT interconnection network [BHQV07] and a complete 64-TCU XMT integer-only chip, both fabricated in 90nm IBM technology.

In [CKTV10], detailed calculations established that using the same generation technology as the GTX 280, a 1024-TCU XMT configuration could be fabricated. We used this configuration to obtain the results in the remainder of this chapter.

5.2.5 Results

Figure 5.3 presents the speedups of all the benchmarks on a 1024-TCU XMT configuration relative to GTX280. Speedups range between $2.05\times$ and $8.10\times$ for highly parallel irregular benchmarks. For one application (BFS), we demonstrate much stronger speedups for limited parallelism, using a dataset with less available parallelism, a synthetic graph with 1M nodes, 3M edges but a diameter of 50,000 Bfs "levels". With this dataset, the average number of active threads per Bfs iteration is 20 (compared to 87.4K threads/iteration above). For this input, the XMT implementation exhibited a speedup of $73.4\times$ over [CBM$^+$09], and $6.89\times$ when compared to a CUDA Bfs implementation for regular, low degree graphs [LWmH10], even when their input processing was not counted. Furthermore, when a

64-TCU XMT configuration was used, the speedup compared to [CBM+09] was 110.6×, the better result explained by the lower latencies in the simpler 64-TCU design, with still enough hardware to handle the problem parallelism.

The two regular benchmarks (Conv and Reduct) show slowdown. This is due to the nature of the code, exhibiting regular patterns that the GPUs are optimized to handle, while the XMT abilities to dynamically handle less predictable execution flow go underused. Moreover, Conv on CUDA uses the specialized Tesla multiply-add instruction, while on XMT two instructions are needed.

Table 5.3 also shows the number of parallel sections executed and the average number of threads per parallel section for each benchmark. For CUDA applications, a large number of parallel sections (kernel launches) implies high overheads caused by switching between CPU and GPU execution and potentially data transfers between the two. A low number of threads also signals inefficient execution for the GPU, since Tesla relies on multi-threading for latency hiding, technique which requires a large number of threads. Note that neither of these factors is an issue for XMT, which has low cost spawn and join primitives, shares cache and main memory between the serial and the parallel processors, and operates well even for low amounts of parallelism.

A more detailed study in [CKTV10] provides a break-down of the execution time spent executing different type of instructions. From that study, we observed that benchmarks with irregular memory access patterns such as Bfs, Spmv and Msort spend a significant amount of their time in memory operations. We believe that the high amount of time spent by Bprop is due to the amount of memory queuing in this benchmark. Conv is highly regular with lots of data reuse, and spends less than half of its time on memory accesses; however, it performs a non-trivial amount of floating-point computation (more than 50% of the remaining time).

The study in [CKTV10] also shows that in the NW benchmark, a significant amount of time is spent idling by the TCUs. From Table 5.3, we observe that the number of threads per parallel section is relatively low in this benchmark. In spite of this high idling time, XMT outperforms the GPU by a factor of 7.36x on this benchmark, illustrating the fact that XMT performs well even on code with relatively low amounts of parallelism. The very large number of parallel sections executed for the NW benchmark (required by the lock-step nature of the dynamic programming algorithm) favors XMT and its low-overhead

synchronization mechanism, and explains the good speedup.

When using a smaller XMT configuration with only 512 TCUs, we observed that the speedup vs. Tesla for the irregular benchmarks was 4.57x on average, while the slowdown was 3.06x for the regular ones. This shows that such an XMT configuration still outperforms the GPU considered, and given XMT's advantage on ease-of-programming, the main point of this comparison holds.

We also note another important outcome of this study. As a rough approximation of the coding effort involved when targeting these two different many-core architectures, we recorded the number of lines of code in each benchmark. This is included in Table 5.3. The significantly lower number of lines of code of the XMT implementations brings supporting evidence to the ease-of-programming claim.

Chapter 6

Application case-study: Maximum-Flow Algorithm on XMT

This chapter describes and evaluates an efficient, scalable implementation of a parallel Max-Flow algorithm for the XMT PRAM-on-chip many-core architecture. We show that by using a PRAM-like architecture, the algorithm is easy to program and efficient, achieving higher speedups than previous work when compared to the best serial implementation. In addition, we compare the XMT Maxflow with a CUDA implementation, and show that the XMT version outperforms the GPU, even when configuring XMT to use approximately the same amount of silicon area as the GPU. This supports the argument that the previous low speedups are not caused by inefficient algorithms or implementations, but because of a mismatch between the algorithm and the underlying platform. It strengthens the case for XMT as an efficient, general-purpose, easy-to-program many-core.

The maximum flow (Max-Flow) problem is a fundamental graph theory problem. A very large number of optimization problems use maximum flow as part of their fastest solution. Outside of network analysis, a short list of applications that use Max-Flow might include airline scheduling, circuit analysis, task distribution in supercomputers, digital image processing, and DNA sequence alignment. There is a high level of interest in efficient parallel solutions to the Max-Flow problem. Recent developments include implementations running on SMPs [BS05, NP11] and GPUs [HH10, STT10, HVD07, VN08].

We describe the Maxflow problem and introduce a number of notations and definitions in Section 6.1. The evolution of serial and parallel Maxflow algorithms is presented in Section 6.2, followed by a brief review the Push-Relabel algorithm in Section 6.3. We then discuss the XMT implementation in Section 6.4. The classes of graphs we used as input are introduced in Section 6.5 and the experimental methodology in Section 6.6. Section 6.7 presents the experimental evaluation results, followed by a discussion in Section 6.8.

6.1 Problem Description

The goal of the Max-Flow algorithm is to determine the maximum flow in a flow network, defined next.

A graph $G = (V, E)$ is a *flow network* if it has two distinguished vertices, a *source* s and a *sink* t, and a positive real-valued *capacity* $c(v, w)$ for each edge $(v, w) \in E$. For simplicity of notation, we extend the capacity function to all vertex pairs by defining $c(v, w) = 0$ if $(v, w) \notin E$.

We refer to the number of vertices $|V|$ as n and number of edges $|E|$ as m.

A *flow* f on a flow network $N = (G, s, t, c)$ is a real-valued function on vertex pairs satisfying the following constraints:

1. Capacity constraint: $f(v, w) \leq c(v, w)$ for all $(v, w) \in V \times V$
2. Anti-symmetry constraint: $f(v, w) = -f(w, v)$ for all $(v, w) \in V \times V$
3. Flow conservation constraint: $\sum_{u \in V} f(u, v) = 0$ for all $v \in V - \{s, t\}$.

The *value* $|f|$ of a flow f is the net flow into the sink:

$$|f| = \sum_{v \in V} f(v, t) \tag{6.1}$$

A *maximum flow* is a flow of maximum value.

The *residual capacity* $r_f(v, w)$ of an edge (v, w) is defined to be $c(v, w) - f(v, w)$. An edge (v, w) is a *residual edge* if $r_f(v, w) > 0$. The *residual graph* $G_f = (V, E_f)$ for a flow f is the graph whose vertex set is V and whose edge set E_f is the set of residual edges.

A directed network $N = (G, s, t, c)$ is a *layered network* if G has the following properties:

1. Each vertex v has a layered number $l(v)$
2. $l(s) = 0$ and $0 \leq l(v) \leq l(t)$ for all $v \in V$
3. If $(u, v) \in E$ then $l(v) - l(u) = 1$

The set $Lj = v : l(v) = j$ is called the jth *layer* of G.

The concept of preflow relaxes the flow conservation constraint, allowing vertices with more incoming flow than outgoing. In this context, we define *flow excess* excess$(v) = \sum_{u \in V} f(u, v)$, the net flow into v.

6.2 Max-Flow Algorithm Background

Serial Max-Flow Algorithms Early solutions to the maximum network flow problem are based on the augmenting path method, which is due to Ford and Fulkerson [FF62]. The Ford-Fulkerson algorithm is pseudo-polynomial in its original form. Edmonds and Karp [EK72] demonstrated that pushing flow along the shortest augmenting path has a polynomial running time of $O(nm^2)$. Dinitz [Din70] suggested searching for augmenting paths in phases and handling all augmenting paths of a given shortest length in one phase, which yields an execution time of $O(n^2m)$. The concept of preflow was introduced by Karzanov in [Kar74], which leads to a $O(n^3)$ algorithm. The execution time has been further improved by using various techniques such as capacity scaling [Gar85] and dynamic trees [GT90].

An alternative method based on the concept of preflow (due to Karzanov [Kar74]) was introduced by Shiloach and Vishkin [SV82]. The SV algorithm reduces the Maxflow problem on a directed graph to $O(n)$ problems on layered networks. Flow is pushed along all shortest paths from the source towards the sink. Excess flow that cannot reach the sink is returned along the same path towards the source. Once all the flow either reaches the sink or is returned to the source, a new layered network is constructed in the residual graph, and the algorithm is re-applied. Designed as a natural parallel algorithm, SV also induces a serial algorithm that is conceptually simple, and has an asymptotic running time of $O(n^3)$.

Goldberg and Tarjan [GT88] replaced the layered network approach by introducing the concept of distance labels in the SV algorithm. Distance labels were easier to manipulate than layered networks and led to more asymptotically efficient algorithms. The GT algorithm maintains a preflow and a distance labeling, and uses push and relabel operations to update the preflow until a maximum flow is found. The label of a vertex is an estimate of the distance to the sink in the residual graph used in the algorithm to guide the push of flows. The raw algorithm is of $O(n^2m)$ complexity. By executing the push and relabel operations in a FIFO order, an $O(n^3)$ algorithm is achieved. The running time of the push-relabel method is improved to $O(nm \log(n^2m))$ in [GT88] by using dynamic trees. The asymptotically fastest solution is presented in [CM98]; it processes participating nodes in descending order of their labels, achieving running time of $O(n^2\sqrt{m})$.

Several algorithms that operate on special type of networks (integer capacities, zero-

Name	Graph	Time	Space	Procs.
Shiloach-Vishkin 82 [SV82]	directed	$T = O(n^2 \log n)$	Space $= O(nm)$	n
Vishkin 86[a]	layered	$T = O(n \log n)$	Space $= O(n^2)$	n
Goldberg-Tarjan 88 [GT88]	directed	$T = O(n^2 \log n)$	Space $= O(m)$	n
Goldberg-Tarjan 89 [GT89]	acyclic	$T = O(n \log n)$	Space $= O(mn)$	m
Vishkin 92 [Vis92]	acyclic	$T = O(n \log n)$	Space $= O(n^2)$	n

[a] Noted as private communication in [GT89]

Table 6.1: Parallel Max-flow algorithms and their complexity bounds.

one networks) and have better asymptotic properties have been proposed. An excellent survey of these developments in maximum network flow problem is presented in [Gol98]. The focus of this study is on algorithms for the most general type of flow networks, and therefore we do not discuss these specialized algorithms any further.

In practice however, the Goldberg-Tarjan algorithm had poor performance, since apparently distance labels were more helpful for improving asymptotic results than implementation runtime. Intuitively, this is because relabel is a local operation, therefore the method loses the global picture of the distances [CG97]. Goldberg's PhD thesis [Gol87] noted the advantage of *global relabels* (in effect, layered networks) that make the implementation closer to the original SV algorithm. The fastest serial implementation that we are aware of (due to Goldberg [Gol06]), includes other heuristics and optimizations, such as *gap relabeling* and *highest-level node selection*.

Parallel Max-Flow Algorithms There have been fewer improvements when it comes to parallel algorithms for Max-Flow. Shiloach and Vishkin [SV82] proposed a first $O(n^2 \log n)$ time and $O(nm)$ space parallel algorithm for directed graphs. Goldberg and Tarjan [GT89] introduced an algorithm for acyclic graphs that runs in $O(n \log n)$ time and $O(nm)$ space. Vishkin [Vis92] extended [SV82] to acyclic graphs and showed an improved $O(n^2)$ space bound that applies to both [SV82, Vis92]. By incorporating the concept of distance labels in the SV algorithm, Goldberg and Tarjan reduced the space requirement down to $O(m)$ [GT88]. The resulting algorithm, called Push-Relabel, is the one used as basis for most subsequent implementations. The chronological evolution of the complexity bounds for the main parallel algorithms is summarized in Table 6.1.

As was the case for the sequential version, implementations of the basic parallel Push-Relabel algorithm were observed to be slow in practice. The convenience of only local

push and relabel operations comes at the cost of poor practical performance [AS92]. This can be addressed by using a number of heuristics, and this was the approach taken by all existing parallel implementations of Maxflow. These are based on the Push-Relabel algorithm enhanced with global relabeling, which is effectively a periodical breadth-first search (BFS) on the residual graph. Alizadeh and Goldberg [AG92] described and implemented a parallel push-relabel algorithm on a Connection Machine CM-2, including global and gap relabeling heuristics. Anderson and Setubal [AS92] evaluated an implementation for a 14-processor SMP, and observed performance improvements of *up to two orders of magnitude due only to global relabeling.* Bader and Sachdeva [BS05] designed a cache-friendly version of the parallel algorithm for a multi-processor, adding a parallel gap-relabel heuristic implementation.

The existing parallel implementations (such as [AS92, BS05]) rely on locks to atomically execute each individual operation in its entirety. Locks, essential for the correctness of these implementations, can lead to contention and increased traffic on the interconnects of parallel architectures, which will affect the scalability of the algorithm. For example, [BS05] reports that an initial implementation of the Anderson and Setubal algorithm [AS92] on a 14-processor shared memory machine actually exhibits slowdowns compared to a sequential implementation. This issue becomes more important when the number of processors is larger, as is the case with emerging many-core architectures.

Several GPU algorithms have been proposed. In [HVD07, VN08], the authors present Maxflow solutions designed for special structured graphs for graphics applications. Push operations have been divided in two phases, push and pull, to avoid lock usages. He and Hong [HH10] evaluate an alternate lock-free, hybrid CPU-GPU implementation designed for general graphs. They show speedups up to 2.5x compared to the fastest sequential code, which is small compared to the peak computing capacity and bandwidth of GPUs. Solomon et. al [STT10] present a GPU implementation based on layered network algorithm for the Max-Flow problem. They do not present speed-ups compared to a best serial implementation, and thus it is difficult to compare the efficiency of their solution.

6.3 Parallel Max-Flow Push-Relabel Algorithm

This section reviews the Maxflow algorithm proposed by Goldberg and Tarjan (GT). The two basic operations of this algorithm are *Push* and *Relabel*, hence this is also known as the *Push-Return* Maxflow algorithm. After discussing a serial algorithm, we review a parallel version which has complexity bounds Time $= O(n^2 \log n)$ and Space $= O(m)$.

The Push-Relabel method is based on Karzanov's concept of Pre-Flow [Kar74], which relaxes the conservation constraint; it allows for the total amount flowing into a vertex to exceed the amount flowing out. Karzanov's algorithm maintains a preflow in an acyclic network; the algorithm pushes flow through the network to find a blocking flow, which determines the acyclic network for the next phase. The Goldberg-Tarjan (GT) algorithm abandons the idea of finding a flow in each phase, and also abandons the notion of global phases.

The GT algorithm maintains a preflow in the original network and pushes local flow excess towards the sink along what it estimates to be shortest paths in the residual graph. This pushing of flow changes the residual graph, and paths to the sink may become saturated. Excess that cannot be moved to the sink is returned to the source, also along estimated shortest paths (which can be different than the path on which the flow reached the vertex). Only at termination does the preflow become a flow, and then it is a maximum flow.

An important issue is how to estimate the distance from a vertex to s or to t. The authors define a *valid labeling* d to be a function from the vertices to the nonnegative integers and infinity, such that $d(s) = n$, $d(t) = 0$ and $d(v) \leq d(w) + 1$ for every residual edge (v, w). The intent is that, if $d(v) < n$, then $d(v)$ is a lower bound on the actual distance from v to t in the residual graph G_f, and if $d(v) \geq n$, then $d(v) - n$ is a lower bound on the actual distance to s in the residual graph.

A vertex v is *active* if $v \in V \setminus \{s, t\}$, $d(v) < \infty$ and excess$(v) > 0$.

Two operations, *Push* and *Relabel* are at the center of the algorithm. These two operations are outlined in Algorithm 6.1. If applicable, *Push* moves excess flow towards what it estimates that are vertices on the path to the sink. If no such neighbors exist, the *Relabel* operation is used to re-evaluate a vertex's distance to the sink by examining its vicinity, and adjusting it accordingly.

Algorithm 6.1 Push and Relabel operations in the GT algorithm

Push(v,w)
Applicability: v is active, $r_f(v,w) > 0$ and $d(v) = d(w) + 1$
Action: Send $\delta = \min(e(v), r_f(v,w))$ units of flow from v to w

Relabel(v)
Applicability: v is active and $\forall w \in V$, $r_f(v,w) > 0 \Rightarrow d(v) \leq d(w)$
Action: Replace $d(v)$ by $\min_{\langle v,w\rangle \in E_f} d(w) + 1$ or by n if $\nexists (v,w) \in E_f$.

The algorithm begins by initializing the preflow f as equal to the edge capacity on each edge emanating from the source and zero on all other edges, and some initial labeling d.

The **serial** algorithm repetitively performs, in any order, the basic operations *Push* and *Relabel* described in Algorithm 6.1. When there are no more active vertices, the algorithm terminates. Proof of correctness, as well as complexity analysis, can be found in [GT88]. As the algorithm allows for quite a bit of flexibility when it comes to the order in which the basic operations are applied, as well as the initial labeling, several versions of this algorithm have been implemented and evaluated. The fastest practical algorithm processes the vertices in descending order of their labels, which leads to the complexity of $O(n^2\sqrt{m})$ [CM98].

A **parallel** Push-Relabel algorithm can be inferred from the serial one by allowing all applicable basic operations to be ran in parallel. However managing the shared data needs to be handled for a correct algorithm. Goldberg and Tarjan [GT88] describe a parallel version of their algorithm which addresses this problem. This presentation is also the basis of my implementation, described in Section 6.4.

The Push-Relabel algorithm proceeds in pulses, each of which consists of a number of operations applied in parallel. Each pulse is divided in four stages: pushing of flow is done during the first stage; relabeling is done in the second; broadcasting new labels is done in the third stage; and finally flow pushed to a vertex in the first stage is added to its excess in the fourth stage. Figure 6.2 outlines these steps.

For an efficient parallel algorithm, computations on binary trees must be performed to allow each vertex to access its incident edges fast. When using a binary tree based data structure, each pulse takes $O(\log n)$ time, and the parallel time of the algorithm is $O(n^2 \log n)$, as shown in [GT88].

Algorithm 6.2 The Pulse operation with the four stages.

Pulse

for all active vertices v **do**
Stage 1 Push flow from v until excess$(v) = 0$ or $\forall w$ such that $d(w) = d(v)-1$, $r_f(v, w) = 0$ (when pushing flow from v to w, reduce excess(v) but do not increase excess(w)
Stage 2 If excess$(v) > 0$ then $d'(v) \leftarrow \min\{d(w) + 1 | r_f(v, w) > 0\}$
Stage 3 If $d(v) \neq d'(v)$ then $d(v) \leftarrow d'(v)$; broadcast $d(v)$ to all neighbors of v
Stage 4 Add flow pushed to v in Stage 1 to excess(v).
end for

6.3.1 Heuristics of Push-Relabel

The Push-Relabel algorithm has been show to be slow in practice. It relies on a number of heuristics to improve its performance. Studies have shown that these heuristics improve running time by up to two orders of magnitude (e.g. [AS93]). Two of the most widely used heuristics are described next.

Global relabeling. The distance labels $d(v)$ for $v \in V$ in the Push-Relabel algorithm represent a lower bound on the distances from any vertex to the sink. These labels help the algorithm to push flow towards the sink as the push operation is always carried from a vertex with a higher label connected to another with a lower label. Global relabeling updates the distance labels on the vertices as the shortest distance from the vertex v to the sink t along the residual graph $G_f = (V, E_f)$. This can be performed by a breadth-first search to the sink. Such a relabeling is performed periodically after a number of push-relabel steps to amortize the expensive computational cost of the heuristic.

Note that with the global relabeling heuristic, the Push-Relabel algorithm becomes more similar to the Shiloach-Vishkin algorithm. The node labels after a global relabel constitute layers, and flow is only allowed to move between layers. However, the Push-Relabel algorithm allows for dynamically relabeling nodes in between global relabels, thus altering the layered network layout in an attempt to direct flow in the right direction when paths towards sink become saturated. This can cause "bouncing" of flow back and forth in some situations, although this is temporarily remedied the next time a global relabeling is ran.

Gap relabeling. This procedure updates the labels of the vertices which are unreachable

from the sink to a label larger than the one of the source, e.g. $n + 1$. Such a situation arises if there are no vertices with labels σ but vertices with distance labels $d(v)$ such that $\sigma < d(v) < n$ exist. The distance labels of such vertices can be updated then to n. Such an update makes it possible to remove these vertices from consideration for pushing flow to the sink at once. The gap relabeling heuristic has been discovered independently by Cherrkasky and by Derigs and Meyer [Che79, DM89].

Several serial implementations use the gap relabeling heuristic, and report improvements, especially when using the highest-label processing order for vertices [CG97]. For parallel implementations, Bader and Sachdeva [BS05] report that a synchronous implementation of the gap-relabel pass (where execution of Push-Relabel operations is stopped while the gap-relabel operation runs) does not improve performance. They introduce a parallel, asynchronous implementation, and show that in some cases, using both global and gap relabeling can improve performance compared with runs with only global relabeling.

6.4 The XMT Max-Flow Implementation xmt_mf

This section introduces our implementation of the Maxflow algorithm using the XMTC platform, as well as various implementation-specific improvements. In [GT88], the authors discuss both parallel and distributed versions of GT. The parallel version is more suitable for shared-memory machines, while the distributed algorithm maps better to message-passing environments. We adapted the parallel GT algorithm for XMT and added several optimizations. We discuss the XMT implementation **xmt_mf** next.

The algorithm proceeds in pulses, each consisting of a number of operations applied in parallel. In each pulse, a sequence of four parallel steps is in parallel. A synchronization point is necessary after each parallel step. In XMT, each parallel step is implemented as a separate `spawn-join` block, which includes an implicit barrier at the join. The four parallel steps are outlined in Algorithm 6.3.

The above description fits the Work-Depth PRAM description, introduced in [SV82]. From this description, we can follow the *XMT Programmer's workflow* discussed in Chapter 3 to derive an efficient and correct implementation in XMTC. We discuss a number of algorithmic and XMT-specific optimizations that we introduced next.

Algorithm 6.3 The four parallel steps in the XMT Maxflow **xmt_mf** Pulse

Push Flow Active nodes push flow on permitted edges. At the destination, incoming flow is stored in a separate location, so that destination excess flow is not yet updated. Flow from multiple incoming edges is accumulated using the atomic, hardware supported prefix-sum to memory XMTC instruction, `psm`.

Relabel Compute Compute new labels for nodes in a separate location. Does not change any node labels to avoid conflicts.

Relabel Update Updates all nodes with new labels computed in the previous step.

Update Flow Updates flow at all nodes with incoming flow computed in *Push Flow* step.

6.4.1 Atomic Updates using Prefix-Sum

The **xmt_mf** implementation is completely lock-free. We used two mechanisms to handle conflicting updates and maintain correctness. First, both the push and relabel operations are split into two phases, one to compute and one to propagate results. Second, the atomic prefix-sum operation is used to coordinate accumulation of flow from multiple sources. The prefix-sum operation is implemented efficiently with decentralized hardware support, allowing performance to scale to tens and hundreds of cores.

Note that in the current implementation of the XMT hardware, only an integer version of the prefix-sum primitive is implemented. However, implementing a floating-point version is feasible, and can be added to a future generation of the architecture. This would allow solving the most general form of Maxflow, which operates with real flows and capacities.

6.4.2 Maintaining the List of Active Nodes

The *Push* and *Relabel* operations are only applicable to vertices that have non-zero excess flow, or *active* vertices. The number of active vertices can be small relative to the size of the graph. For example, [HH10] observes that for a graph of with 262,144 vertices and 1,276,928 edges, only a few hundred vertices are typically active, briefly peaking at a maximum of 1,400. This is about 0.5% of the total number of vertices. However, in their implementation, they start one thread for each graph vertex, even though only a small fraction of the threads will actually contribute useful work.

In the **xmt_mf** implementation, our implementation maintains two lists of vertices:

current active vertices This is a list of all the vertices that are active in the current pulse, i.e. they have non-zero excess flow at the beginning of the pulse.

updated vertices This is a list of all the vertices that changed their excess flow during a pulse, resulting in a non-zero excess value.

In the *Push Flow*, *Relabel Compute*, *Relabel Update* steps, only the vertices in the *current active vertices* participate. In the *Update Flow* step, only the vertices in the *updated vertices* list participate.

At the end of the pulse, the vertices in the *Updated vertices* become the *current active vertices* list for the next pulse. This is done efficiently by a simple pointer swap in XMTC.

Maintaining unique entries the *updated vertices* list is done using a **gatekeeper** mechanism. A thread that discovers non-zero excess vertex v first executes a prefix-sum to memory (psm) operation on a memory location gatekeeper(v). Only if the value returned by the prefix-sum operation was 0 (i.e. the thread was the first to attempt this), the thread is allowed to add vertex v to the list.

The queuing factor at the gatekeeper(v) location is bounded by the in-degree of the vertex v. In practice this factor is fairly low, since threads in XMT are allowed to execute at their own speed, and the likelihood that the neighbors of v attempt to push flow towards v at the same moment in time. In the experimental evaluation, where we tested a number of different classes of input graphs, we did not observe the queuing factor at the gatekeeper to be an issue.

Adding an element to the *updated vertices* list also requires a prefix-sum operation. The number of elements in the list is first increased using a prefix-sum to register operation (ps). The value returned by ps is used as the index in an array for storing the vertex number. By using the constant-time `ps` operation, fast execution is guaranteed even when a large number of threads attempts to add vertices to the list in parallel.

6.4.3 Implementation of Global Relabeling

We implemented the Global Relabeling heuristic by a parallel Breadth First Search algorithm in the residual graph. Instead of using BFS levels, we update the labeling $d(v)$ during the breadth-first traversal of the graph.

We used two BFS passes:

- A *backwards* BFS run starting from the sink. In a backward BFS, an edge $v \rightarrow w$ exists iff $r_f(w, v) > 0$. In other words, if an anti-parallel edge $w \rightarrow v$ exists and has non-zero residual capacity, then the edge $v \rightarrow w$ is part of the BFS tree.

  The sink t has the label 0. Vertices v such that $r_f(v, t) > 0$ will be labeled $d(v) = 1$ etc.

- A *forward* BFS run starting from the source. In this pass, only the vertices that have not been labeled in the backwards pass are considered. These are the set of vertices from which the sink is not reachable in the residual graph, and hence any excess flow should be pushed towards the source instead. To accomplish that, these vertices are relabeled with labels $d(v) > n$.

6.4.4 Parallel Breadth-First Search

The global relabeling pass used in most Push-Relabel implementations requires a breadth-first traversal of the residual graph (Breadth-First Search, or BFS). For XMT, we used a parallel implementation of BFS.

BFS is regarded as an algorithm that is difficult to parallelize efficiently. It exhibits irregular execution and memory access patterns, as well as varying degrees of parallelism. However, previous work has shown that XMT implementations of BFS are significantly more efficient than comparable parallel solutions [CKTV10].

A parallel BFS algorithm presented in a Work-Depth framework is shown in Algorithm 6.4.

Algorithm 6.4 Breadth-First Search Work-Depth Algorithm

```
1:  label_0 ← {source}
2:  level ← 0
3:  while label_level ≠ Φ do
4:    for all v ∈ label_level do
5:      for all w such that (v, w) ∈ E do
6:        if w not visited then
7:          mark w as visited
8:          add w to label_level+1
9:        end if
10:     end for
11:   end for
12:   level ← level + 1
13: end while
```

In Section 3.3.3 several variants of BFS were compared by using the LSRTM modeling.

By considering the current XMT embodiments' capabilities, as well as the characteristics of the inputs used, we chose a serialized inner spawn implementation of BFS. This is derived from Algorithm 6.4. The main change is that the inner parallel loop in lines 5-10 is implemented using a serial loop. Note than when fully optimized nested parallelism becomes available in the XMTC compiler (see e.g. [TCBV10]), a nested parallel BFS could be used instead, with potential performance benefits.

To ensure the atomicity of the operations in lines 6-8, a gatekeeper array is used, where a prefix-sum instruction to a memory location is performed before processing a node. Only the first thread encountering an unvisited node proceeds to visit and mark it.

6.4.5 Gap Relabeling

In [BS05], the authors implement and evaluate an asynchronous gap relabeling pass: this is carried out concurrently with the push/relabel operations. Implementing an asynchronous gap relabeling heuristic implies a departure from the Work-Depth model, and it would involve a non-trivial implementation effort in a PRAM-like program model such as XMT.

We experimented with a synchronous gap relabeling optimization. After a fixed number of parallel pulses, a gap relabeling pass is executed, after which execution of pulses resumes.

To implement the heuristic, first one thread per vertex is started to mark which labels are used and which ones are not. Next, the smallest unmarked (i.e. unused) label σ is identified using a balanced-binary tree minimum finding function. And finally, another parallel step is used to relabel all vertices with labels $\sigma < d(v) < n$ with $d(v) \leftarrow n + 1$. This marks that the sink is not reachable from those nodes, and they should redirect their excess flow towards the source.

We evaluated the synchronous gap relabel implementation on all the input graphs considered with various parameters. We did not observe any improvements over using the gap relabel implementation, and some cases actually encountered slow-downs. Our conclusion is that this optimization is not applicable to the current implementation. We are not discussing the gap relabel heuristic in the remainder of this work.

6.5 Structure of Input Graphs

The practical performance of the MaxFlow parallel implementations depends on the structure of the graph. This is not captured by the asymptotic complexity, but has a significant effect in practice. The speedups that can be achieved are upper bounded by the degree of parallelism available at execution time, which is given by the number of *active* nodes at each step. This is intrinsically related to the structure of the graph, and can vary largely even for graphs with similar number of nodes and edges. For example, a graph consisting of a very "long" chain of layers will likely execute a large number of steps with few active nodes per step. In contrast, a dense graph with a small diameter will usually execute fewer pulses but many active nodes per step.

In addition, the number of pulses also determines the amount of synchronization necessary at runtime. Since a barrier is necessary after each step, graphs that do require many steps tend to cause inefficient executions on platforms where the cost of synchronization is high.

To quantify the effect of the structure of the graph on performance, we used a number of input graph with various layouts. A set of synthetic graph families was proposed as part of the First DIMACS challenge [JM93]: Washington random level (RLG), RMF, Acyclic Dense (ADG). We describe these below. Instances of these families of graphs have been used in most research papers evaluating MaxFlow implementations, and they have become somewhat of a standard.

In addition to the DIMACS graph families, we use an additional type of graphs for the performance evaluation: Random. As opposed to the more structured graphs in the DIMACS challenge, this type of graph has a more irregular structure, exercising different characteristics during execution of the MaxFlow implementation.

The families of input graphs generated and used for the evaluation are discussed next.

GenRMF graphs These are graphs proposed for the First DIMACS Challenge [JM93]. They are created using the generator based on RMFGEN of Goldfarb and Grigoriadis [GG88]. These graphs are made of l_1 square grids of vertices (frames), having $l_2 \times l_2$ vertices, and connected to each other in sequence. The source is in a corner of the first frame, and the sink is in the corner of the last frame. Each vertex is connected to its grid neighbors within the frame, and to one vertex randomly chosen from the

next frame. There are about $6n$ edges in the network. Edge capacities within a frame are $10^4 \times l_2 \times l_2$. Capacities for edges between frames are chosen uniformly from the range $[1, 10^4]$. Two sub-classes were considered: *wide*, with few and large frames, and *long*, with many and small frames.

RLG Graphs There are also graphs used for the First DIMACS Challenge. The generator used is WASHINGTON, developed by Anderson et al. These are rectangular grids of vertices, where every vertex in a row has 3 edges to randomly chosen vertices in the following row. The source and the sink are external to the grid: the source has edges to all vertices in the top row, and all vertices in the bottom row have edges to the sink. Edge capacities are integers drawn randomly and uniformly from $[1, 10^4]$. Two sub-classes were considered: *wide*, in which there are more columns than rows; and *long*, with more rows than columns.

ADG Graphs These graphs were also part of the First DIMACS Challenge. Generated using the AC generator by Setubal et. al. These are complete, directed acyclic graphs. Edge capacities are in the range $[1, 10^6]$.

Random Graphs Edges are placed between pairs of nodes chosen uniformly at random. Because of this structure, the graphs often have very short diameter, and offer lots of parallelism. We wrote a generator for graphs in this class. By placing the edges uniformly at random between pairs of nodes, the resulting graph has relatively uniform average node degrees.

For the DIMACS graph families, we used the synthetic generators available from `http://www.avglab.com/andrew/CATS/maxflow_synthetic.htm`.

6.6 Experimental Methodology

We evaluated the performance of our parallel Max-Flow implementation by comparing to the running time of the fastest known serial code **hi_pr** [Gol06]. In addition, we also compared parallel algorithm performance using a recent CUDA implementation running on NVIDIA GPUs. We describe the platforms and methodology of obtaining the performance results for each implementation below.

CPU	AMD Phenom 9600B
Clock frequency	2.3GHz
Number of cores	4
L2 cache	4x512MB
L3 cache	2MB
DRAM	4GB DDR2-SDRAM

Table 6.2: Specifications of serial evaluation platform

6.6.1 Serial Experiment

We evaluated the execution time of the serial Maxflow implementation using two platforms: (i) a modern commercial x86 architecture and (ii) the Master TCU of the Paraleap FPGA. We describe the two experiments to collect the serial execution time next.

6.6.1.1 Running on x86 CPU

Based on a review of the literature, we identified the **hi_pr** code as being the fastest serial MaxFlow implementation available [Gol06, HH10]. This implementation uses the highest label node processing order, as well as both the global relabeling and gap relabeling heuristics.

We executed this implementation on a x86 CPU, namely an AMD Phenom 9600B at 2.3GHz. The full specifications of the evaluation platform are shown in Table 6.2. Note that even though the processor used has 4 computing cores, the program run is sequential, and thus uses only one core. However, the L3 cache is shared between all the cores, and hence each core has access to all the cache.

To reduce system interferences and variability, we ran each input 100 times, and computed the average cycle count by dividing the total time by the number of executions (i.e. 100). To increase the accuracy of measurement, we used the *rdtsc* instruction was used to get timing information which reports clock cycles elapsed. The AMD processor used supports the `nonstop_tsc` CPU capability which guarantees that the timestamp counter *tsc* is incremented every clock cycle.

6.6.1.2 Running on Paraleap Master TCU

We ported the **hi_pr** code to run on the Master TCU core of the Paraleap FPGA prototype. Since the MTCU is a general-purpose serial processor which includes an L1 cache,

GPU	GTX 480
CUDA cores	480
Processor clock	1.4GHz
DRAM	1536MB DDR5
DRAM clock	1.84GHz

Table 6.3: Specification of the NVIDIA GPU used for the CUDA experiment

no additional optimizations are needed for an efficient execution.

The only change that was necessary for the **hi_pr** code was handling of the input data. Since Paraleap does not support file I/O at this time, the input data has to be pre-loaded into DRAM memory before execution. Note that reading the data from disk is not timed in any of the experiments (serial or parallel), and therefore the way the data is loaded from disk does not have any impact on the relative performance of the implementations or platforms.

6.6.2 XMT Parallel Experiment

The parallel implementation was run on two different embodiments of the XMT architecture: (i) the 64-core Paraleap prototype, built using FPGA technology; the cycle counts reported by Paraleap have been shown to reflect a much faster ASIC [WV08a] (e.g. 800MHz or higher) and (ii) a forward-looking 1024-core configuration simulated using XMTSim. Previous work [CKTV10] has shown that this configuration, if built using same technology as today's GPUs, would use approximately the same area as an NVIDIA GTX280 chip, and could run at similar clock speed.

The execution time was measured by reading the timestamp counter on both XMT-Sim and Paraleap. This is an accurate counter incremented at each clock cycle, and can therefore be used to collect exact execution cycle counts.

6.6.3 GPU Parallel Experiment

We also evaluated a CUDA implementation of Maxflow using an NVIDIA GPU. For this experiment, we used a NVIDIA GTX 480 GPU, based on the Fermi architecture. The characteristics of the GPU are included in Table 6.3. We used the CUDA Toolkit v3.2 to compile and execute the CUDA programs on the GPU.

The card is part of a system used for this experiment has a dual core AMD Opteron

2218 at 2.67GHz, with 2x1MB of cache and 4GB or DDR2 DRAM.

The execution time is collected using accurate timer functionality on the CPU. By using the CPU timer, this ensures that both the GPU and supporting CPU execution time is measured.

To evaluate the performance of Maxflow on a GPU, we used an implementation based on the work of Heng and Ho [HH10]. In the referenced paper, the authors describe a hybrid CPU-GPU algorithm **hybrid_mf**, which switches execution between the CPU and the GPU dynamically, in order to maximize performance. The **hybrid_mf** algorithm makes the decision to switch execution to the GPU only when enough active nodes (and thus the degree of execution parallelism) exist to justify the overheads of transferring the data and control to the GPU. If the number of active nodes falls under a certain threshold, execution is switched back to the CPU.

The threshold to switch from CPU to GPU execution was determined to be relatively high in [HH10]. For the configuration considered in their paper, a number of about 4,000 active nodes is required. For the input sizes we were able to run, this threshold is not reached, and the **hybrid_mf** algorithm is always using the CPU implementation.

Instead of using the **hybrid_mf** algorithm, we evaluated the performance of the CUDA-only implementation **cuda_mf**, also described in [HH10]. By using the **cuda_mf** we ensured that the GPU is used for execution for the entire algorithm. Note that the speed-up results presented in [HH10] are obtained using the **hybrid_mf** implementation.

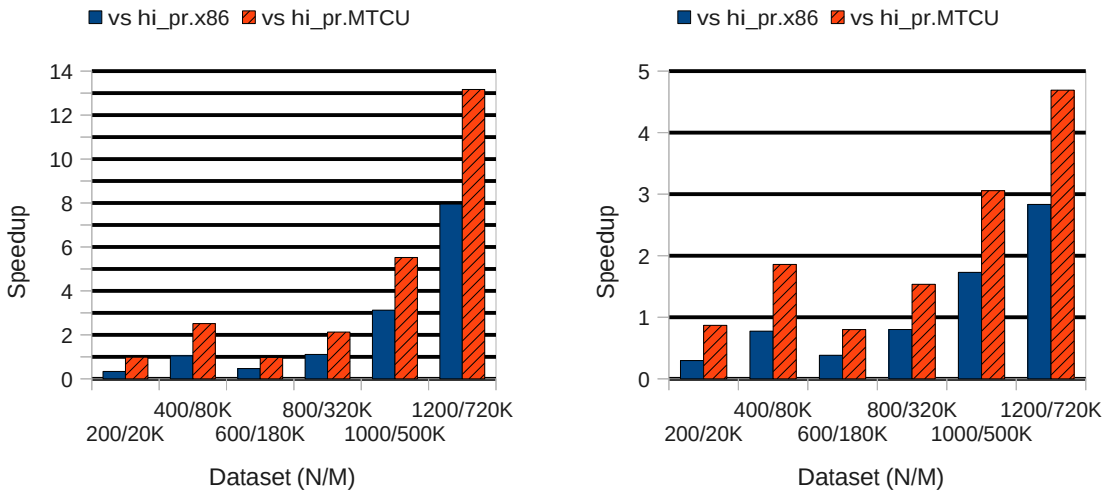
6.7 Performance Evaluation

This section presents performance results for the XMT implementation of MaxFlow. We show speedup results compared to a best serial solution running on a serial processor, followed by a comparison with a CUDA parallel implementation running on an NVIDIA GPU.

6.7.1 Speedups vs. Serial Max-Flow

We executed the XMT Maxflow implementation using the two configurations described in Section 6.6.2 and input graphs in all the classes described in Section 6.5.

Figure 6.1 shows the speedup results of the **xmt_mf** implementation compared to the



(a) Speedup of **xmt_mf** on XMTSim/1024

(b) Speedup of **xmt_mf** on Paraleap/64

Figure 6.1: Speedups vs. serial for **xmt_mf** on Acyclic Dense Graphs (ADG)

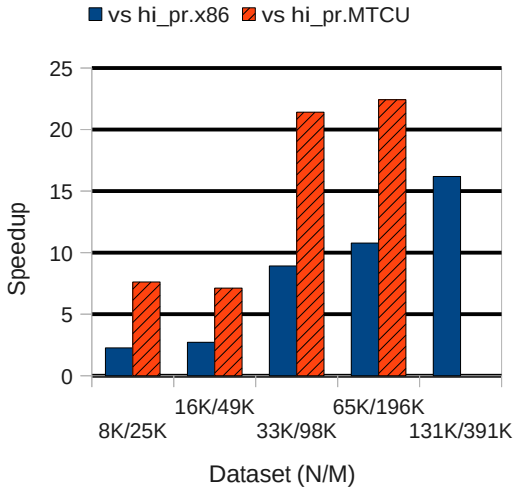
hi_pr serial implementation. The inputs are Acyclic Dense Graphs, with sizes varying from 200 nodes and 20,000 edges to 1,200 nodes and 720,000 edges. Figure 6.1a shows the speedups achieved by simulating the **xmt_mf** implementation using XMTSim on a configuration with 1024 TCUs. Figure 6.1b shows the speedups when running the same implementation on the Paraleap FPGA prototype, configured with 64 TCUs.

We observe that for each class of input graphs, the speedup results tend to get higher as the size of the input grows. For the two graphs (400/80K) and (600/180k), the reason that the speedup is lower is because the serial implementation **hi_pr** was able to take advantage of the gap relabeling heuristic.

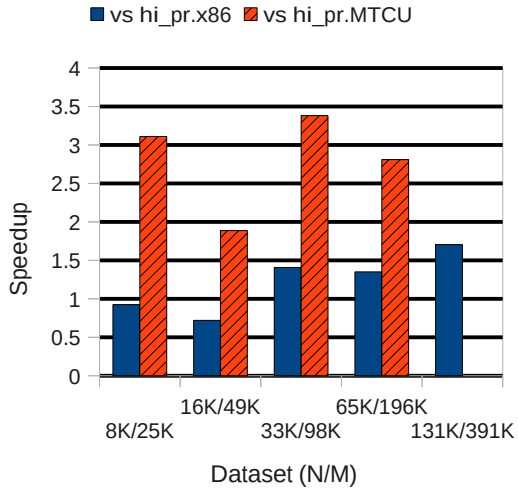
Figure 6.2 shows the speedups for the same implementations and platforms but with Washington Random Level Graphs (RLG) as input. The generated graph sizes vary from 8,000 nodes and 25,000 edges to 131,000 nodes and 391,000 edges.

In Figure 6.3 we show the speedup results when using Random graphs (RAND) as input. The average node degree is 6, and the generated graph sizes vary from 10,000 nodes and 60,000 edges to 65,000 nodes and 393,000 edges.

The speedup results for the RAND graphs running on the XMTSim/1024 configuration are particularly encouraging. Because of the way in which they are generated (adding edges between pairs of nodes chosen uniformly at random), these graphs have short diam-

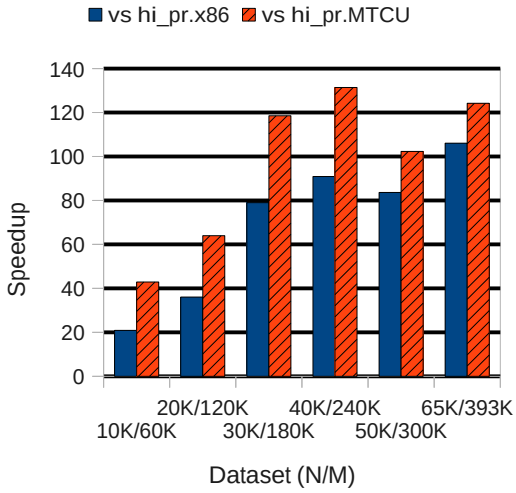


(a) Speedup of **xmt_mf** on XMTSim/1024

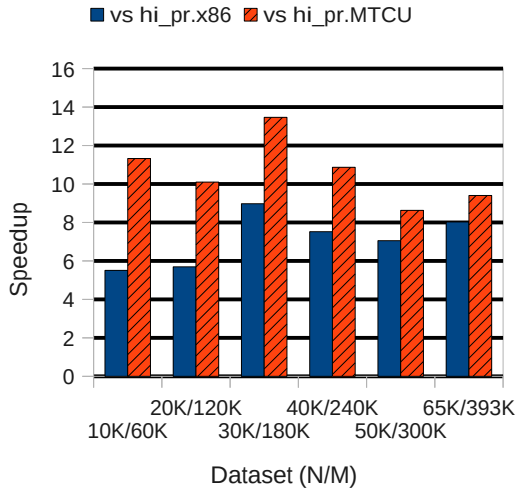


(b) Speedup of **xmt_mf** on Paraleap/64

Figure 6.2: Speedups vs. serial for **xmt_mf** on Washington Random Level Graphs (RLG)

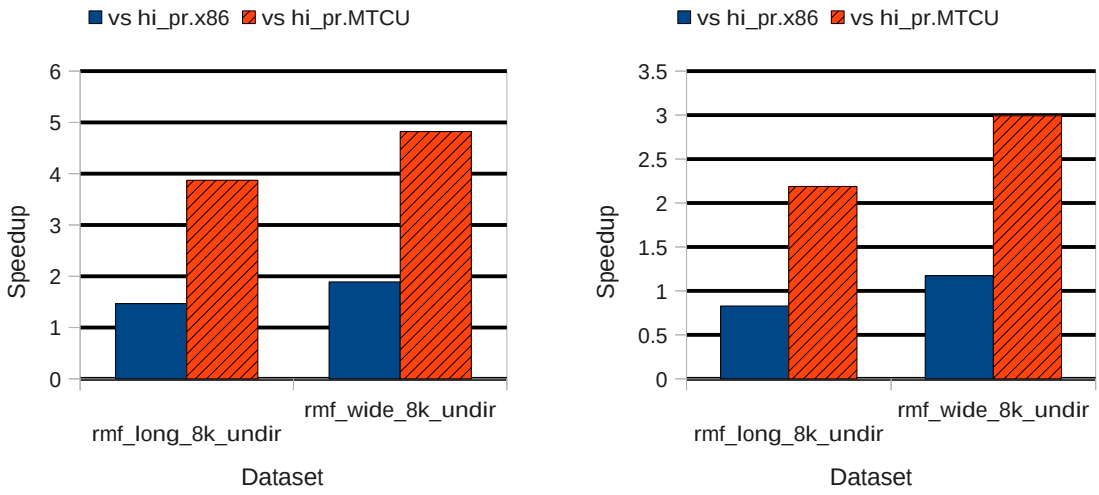


(a) Speedup of **xmt_mf** on XMTSim/1024



(b) Speedup of **xmt_mf** on Paraleap/64

Figure 6.3: Speedups vs. serial for **xmt_mf** on Random Graphs (RAND)



(a) Speedup of **xmt_mf** on XMTSim/1024

(b) Speedup of **xmt_mf** on Paraleap/64

Figure 6.4: Speedups vs. serial for **xmt_mf** on GenRMF graphs

eters and provide high level of parallelism for the MaxFlow implementation. This allows higher utilization of the XMT cores, leading to the higher speedup.

When running on the Paraleap/64 configuration, the hardware reaches full utilization even for smaller graphs, causing the "flattening" of the speedup observed in Figure 6.3b.

Figure 6.4 shows the speedups when using GenRMF graphs as input. Two input graphs were used: one of the "wide" type, with 8192 nodes and 45,000 edges, and one of the "long" type, also with 8192 nodes and 46,000 edges.

6.7.2 Comparison with GPU MaxFlow

Figures 6.5-6.7 present the results of comparing the **cuda_mf** execution time with the XMT Maxflow implementation.

For the XMT execution, we used the 1024-TCU configuration simulated using XMTSim. We have shown in [CKTV10] that such a configuration uses roughly the same amount of silicon area as a NVIDIA GTX 280 GPU.

Figure 6.5 shows the speedup of **xmt_mf** vs. **cuda_mf** using the same Acyclic Dense Graph (ADG) inputs as in Section 6.7.1, with sizes varying from 200 nodes and 20,000 edges to 1,200 nodes and 720,000 edges. In Figure 6.6 we present the speedups when using RLG input graphs with sizes varying from 8,000 nodes and 25,000 edges to 131,000 nodes

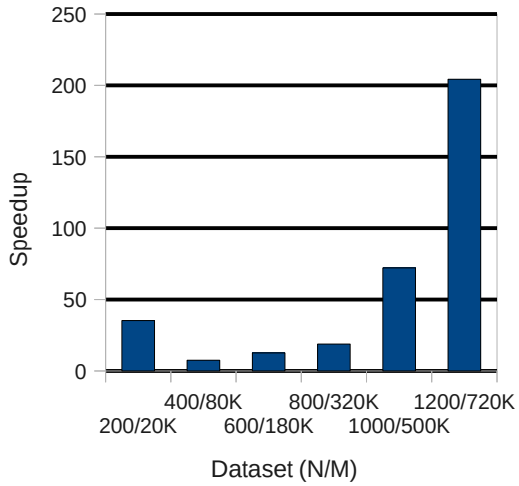


Figure 6.5: Speedup for **xmt_mf** compared to adf_pure_cuda on NVIDIA GTX480 Fermi GPU on Acyclic Dense Graphs (ADG)

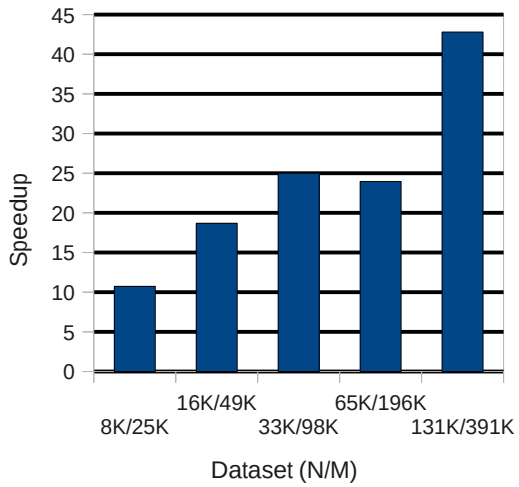


Figure 6.6: Speedup for **xmt_mf** compared to adf_pure_cuda on NVIDIA GTX480 Fermi GPU on Washington Random Level Graphs (RLG)

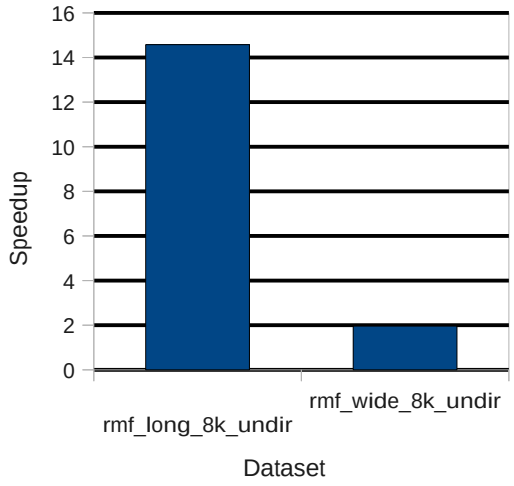


Figure 6.7: Speedup for **xmt_mf** compared to adf_pure_cuda on NVIDIA GTX480 Fermi GPU on GenRMF graphs

and 391,000 edges. Finally, Figure 6.7 displays the speedup results on the GenRMF "wide" graph, with 8192 nodes and 45,000 edges, and the "long" type, also with 8192 nodes and 46,000 edges.

6.8 Discussion

In this chapter we described and evaluated an XMT implementation of the Push-Relabel Maxflow algorithm by comparing execution time to both (i) a best serial implementation running on a modern serial architecture and (ii) a parallel implementation running on a GPU.

A similar approach to the four-step description of the Pulse operation in **xmt_mf** (Algorithm 6.3 was described by Hussein et. al in [HVD07]. They present an implementation of the Push-Relabel algorithm targeted only at grid graphs. They split the *Push* operations into two parallel steps, push and pull. However, they can ensure consistency of relabeling by taking advantage of the grid structure of the graphs, and they only need one parallel step for that. Since we are targeting general graphs, we need to split *Relabel* as well to avoid conflicting updates and maintain a correct labeling of vertices.

When comparing the 1024-TCU XMT configuration to the AMD Phenom serial processor, the largest speedups are 7.95x for the ADG graphs, 16.19x for RLG graphs, 108.3x for

RANDOM graph and 1.89x for GenRMF graphs, as shown by Figures 6.1a, 6.2a, 6.3a and 6.4a respectively. Overall, the results show improved performance across all the classes of graphs, and there is potential for even higher speedups for larger datasets that provide more parallelism.

For the comparison of **xmt_mf** with the GPU **cuda_mf** implementation, the largest speedups are 204.31x for the ADG graphs, 42.8x for RLG graphs and 14.58x for the GenRMF inputs, reflected by Figures 6.5, 6.6 and 6.7 respectively. This reflects the difficulty of the GPU programming model to scale down and provide performance when the amount of parallelism is relatively low. The GPUs provide very good performance for certain types of workloads (regular and predictable memory accesses, very high amount of parallelism, limited synchronization), but fall behind on many others, such as Maxflow or other graph applications.

One of the main challenges of the field of parallel computing is to build a system that can cope with any amount of parallelism without large performance degradation. Using the Maxflow example, we provided some initial evidence that XMT has the potential to address this challenge.

Chapter 7

Conclusions and Future Work

7.1 Summary of Contributions

Faced with nearly stagnant clock speed advances, chip manufacturers have turned to parallelism as the source for continuing performance improvements. But even as parallel architectures have flooded the market, a universally accepted methodology for programming these for general purpose applications has yet to emerge. Existing solutions tend to be hardware-specific, rendering them difficult to use for the majority of application programmers and domain experts, and not providing scalability guarantees for future generations of the hardware. This gives current programming approaches for parallel on-chip architectures the characteristics of a "moving target". This open problem is currently the main stumbling block for the industry in getting the upcoming generation of multi-core architectures to improve single task completion time using easy-to-program frameworks.

The eXplicit Multi-Threading (XMT) project developed at University of Maryland attempts to address this impasse. By relying on an established algorithmic model and a high-level abstraction, XMT decouples the programmer from the hardware implementation. To deliver on the ease of programming, performance and scalability goals the XMT platform promises, more than just the hardware is required. This thesis advances towards building the environment needed for application designers interested in writing efficient and scalable parallel programs for XMT. More concretely, the contributions of this thesis can be summarized as follows:

- We presented a workflow guiding programmers to produce efficient parallel solutions starting from a high-level problem. We discussed each of the stages of the process, introducing models with increasing levels of specificity and used the graph breadth-first search problem as a running example.

- We introduced an analytical performance model for parallel PRAM-based programs by defining the Computation Depth, Length of Sequence of Round-Trips to Mem-

ory (LSRTM) and Queuing Delay (QD) metrics, providing a methodology to derive them from an XMTC program and inferring formulas to estimate program running time based on these values. We illustrated the approach by applying it to three computational problems: summation, prefix-sums and graph breadth-first search. We established the validity of the model by comparing relative performance of different algorithms for each problem, and confirming the results empirically by simulation.

- We presented RAP – an improved compiler loop prefetching algorithm targeted at many-core architectures, and evaluated it on XMT. We showed that under resource constrained scenarios it outperforms Mowry's loop prefetching algorithm by up to 40.15%, the GCC improved implementation by up to 34.79% and a simple hardware prefetching scheme by up to 24.61% on average over our benchmark suite. The RAP algorithm is robust, providing considerable improvements and never falling behind significantly on any of the hardware configurations tested, making it a timely and necessary addition to compilers targeting fine-grained many-core architectures. In addition, we conducted a design-space exploration focused on resources directly affecting support for memory-level parallelism on XMT. We identified the Pareto-optimal hardware-software configuration which delivered 53.66% performance improvement on average while using only 5.47% more chip area than the bare-bones design. The RAP algorithm was key to objectively evaluating each design point by adapting to the prefetch resources available.

- By implementing a comprehensive set of parallel benchmarks, we compared the performance of XMT with two modern processors: an Intel Core 2 Duo CPU programmed using a serial paradigm (but using a parallelizing compiler), and a NVIDIA GTX280 GPU programmed using the CUDA framework. We showed that when using a respective equivalent configuration, XMT provided greatly improved or competitive performance with both the CPU and the GPU. When comparing a 64-TCU XMT configuration to the Intel CPU, we observed speedups ranging between 6.3x and 13.89x for the data sets that fit in the cache, and 2.5x to 8.18x for data sets exceeding the size of the cache on both platforms. Compared to the GPU, a 1024-TCU configuration provided speedups of 2.05x to 8.10x on applications with irregular memory access patterns, and slowdowns of 0.23x to 0.74x on regular benchmarks. The

latter results show that even though GPUs are optimized for these kind of applications, XMT does not fall behind significantly, not an unreasonable price to pay for ease of programming and programmer's productivity.

- We reviewed an XMT many-core implementation of the Max-Flow algorithm and its evaluation. Although other implementations could not achieve speedups in excess of 2.5x versus a best serial algorithm (**hi_pr**) on current many-cores, we demonstrated a potential for much better performance on XMT. This example provides powerful new evidence that XMT is better suited to handle general-purpose, irregular applications.

7.2 Peer Reviewed Publications

The majority of the the material in this thesis has been included in peer-reviewed publications. The programmer's workflow and performance model described in Chapter 3 appeared in [VCL07]. The RAP algorithm and the design-space exploration study of Chapter 4 were presented in [CTK+10, CTK+11]. The evaluation from Chapter 5, comparing XMT performance with existing serial and many-core architectures was published in [CSWV09] and [CKTV10] respectively. The evaluation of an XMT MaxFlow algorithm in Chapter 6 expands upon preliminary results appearing in [CV11].

7.3 Directions for Future Work

We discuss some ideas for future work which build upon the contributions of this thesis.

- Make other prefetching optimizations resource-aware. For example, work on making the linear prefetching also resource aware.
- Provide a library of optimized parallel primitives. Summation, prefix-sums and potentially others.
- More comprehensive MaxFlow analysis. For example, compare with other parallel algorithms.

7.3.1 Automatic or Assisted Performance Modeling

In Chapter 3 we introduced an analytical performance model based on metrics such as LSRTM and QD. We showed how to compute these metrics by examining a description of the implementation either in the Work-Depth or XMT Programming model.

Extracting these metrics manually can be tedious and error prone. An alternative approach would be to build a tool that can take as input an XMTC implementation, and extract such information from it. Where the tool cannot make an exact estimation, such as the Queuing Delay for a memory location accessed through an ambigous pointer derefence, it can ask for the assistance of the programmer. Alternatively, the programmer can provide annotations to the code, transferring her knowledge of the semantics of the application to the modeling tool.

7.3.2 Compiler Support for Read-Only-Buffers (ROBs)

The Read-Only Buffers are memories located at cluster level that can be used to cache and reuse data read by the TCUs in the same cluster. However, they cannot be used transparently by the TCUs as distributed caches. The main difference from caches is that the Read-Only buffers are not maintained coherent, meaning that there is no hardware mechanism to ensure that the data stored in one buffer reflects the actual values stored in the shared memory, or in a different buffer. This can potentially cause correctness problems, if stale data is read from the ROBs by a program, but the programmer expects it to be consistent.

To avoid such problems, the compiler controls what data is stored in the ROBs. The compiler can choose between two types of read instructions for a particular instruction: a regular and a cacheable load, as described in Section 2.3. Moreover, since the ROBs have limited capacity, the compiler should limit the data that is stored in the ROBs only to the data that will be re-used from within the same cluster.

Future work could add an analysis pass in the XMTC compiler to identify data that is both safe and beneficial to be stored in the Read-Only-Buffers. The pass will start by identifying data that is read-only for all the threads and that is also read by more than one thread. As an example of such data, consider an application that uses a constant “mask” array common to all threads. This mask array is a good candidate to be stored in the

Read-Only buffers.

Note that a clear decision about the read-only status of data cannot always be made. For example, when pointers are used in an XMTC program, and the compiler analysis cannot determine conclusively if there are any aliases for a particular memory location. In these cases, the compiler will make the conservative decision not to use the Read-Only Buffers for the affected addresses. This will ensure program correctness, at the possible cost of slight performance loss.

7.3.3 Library-Enhanced Parallel Programming

A common practice aiming at increasing usability of systems is to provide libraries with optimized versions of commonly used operations, which can then be used as building blocks in more complex applications. Recent such efforts targeting parallel architectures include the CUDA Thrust library for NVIDIA GPUs [HB09] and the PAD library for the SB-PRAM platform [KKT01].

The XMT platform could definitely benefit from such a library. The parallel summation and prefix-sums algorithms discussed in Chapter 3 constitute clear candidates for such a library.

More complex algorithms such as Breadth-First Search could also be added, assuming the data structures used were designed to accomodate a wider range of input data types. Note that one application to use a BFS library function is the MaxFlow implementation discussed in Section 6, along with other parallel graph algorithms such as Bi-connectivity.

7.3.4 Improved Benchmark Suite

Benchmarking on-chip parallel architectures such as emerging multi- and many-cores is a relatively new area. While serial architectures as well as large, distributed systems have established benchmarking suites (e.g. SPEC, LINPACK), there is no such widely used standard for many-cores.

To establish the competitiveness of XMT, more benchmarks need to be implemented in XMTC to reflect more application domains and workload types. For example, more of the benchmarks in the recent Rodinia benchmark suite [CBM+09] should be ported for XMT. The performance evaluation discussed in Chapter 5 includes three out of the nine kernels

in Rodinia: Back Propagation, Needleman-Wunsch and Breadth-First search. The remaining six are all good candidates for XMT implementations: K-means, HotSpot, Leukocyte Tracking, Speckle Reducing Anisotrpoic Diffusion (SRAD), Stream Cluster and Similarity Scores. Another resource can be the benchmarks used by Williams et. al [WWP09]: Sparse Matrix-Vector multiply (SpMV), Lattice-Boltzmann Magnetohydro-dynamics (LBMHD), 3D Stencil and 3D FFT, with which we have some overlap as well. Another effort that has been getting attention in the research community is based on the so-called "Berkeley Dwarfs" [ABC$^+$06, ABD$^+$09], an attempt to capture the most commonly used programming patterns used across all application domains. While implementations have started to appear, it is not clear yet what level of adoption they will see as parallel benchmarks.

With more parallel architectures being brought to market, additional benchmark suites are certain to be proposed. To the extent possible, it is important to provide corresponding XMT implementations and results. Such studies provide an intuitive method to compare and contrast the decisions made in designing different parallel solutions.

Appendix A

XMTC Code

A.1 K-ary Tree Summation

```
/*
 * void sum(...)
 *
 * The function computes sums by using a k-ary tree.
 * k is defined by the parameter k to the function.
 *
 * Input:
 * increment[] - an array of increment values
 * k - the value of k to use for the k-ary tree
 * size - the size of the increment[] array
 *
 * Output:
 * result[] - element 0 of the array is filled with the sum
 *
 */
void sum(int increment[], int result[], int k, int size) {
        register int low, high;
        int height = 20; //note: height should be as large as log_k(size)
        //int layersize[height]; // number of nodes in layer i
        int base[height*size]; // base of leaf is its value after PS
                                // base of internal node is the base of its leftmost leaf
        int sum[height*size]; // the value of sum for a node is the sum of the values of
                               // increment for all its leaves

        int iteration = 0; // determines the current height in the tree

        int temp;
        int done; //a loop control variable

        int l,  //level where leaves start
                sb,     //index where leaves would start if size is a power of 2
                d,      //size - k^l
                offset, //how much to offset due to size not being power of 2
                sr,             //sb + offset
                over,   //number of leaves at level l
                under,  //number of leaves at level l + 1
                sbp1;   //index of one level higher from sb
        int fill; //nodes to fill in with 0 to make all nodes have k children

        int level, startindex, layersize;

        int i;

        /*
         * With non-blocking writes 0 RTM is required to initialize
         * the function parameters: k and size
         * 0 RTM is required to initialize local variables such as height
         */

        //Special case if size == 1
        if(size == 1) { //the check has 0 RTM because size is cached.
                result[0] = 0;
                return;
        }
```

```
/*
 * 18 lines of code above, means computation cost = 18 up to this point.
 */

//calculate location for leaves in the complete representation
l = log(size) / log(k);

sb = (pow(k,l) - 1) / (k - 1); //this is derived from geometric series
sbp1 = (pow(k,l+1) - 1) / (k - 1);

d = size - pow(k,l);
offset = CEIL(((double) d) / (k - 1));
sr = sb + offset;
over = pow(k,l) - offset;
under = size - over;

/*
 * Computation cost = 8
 */

//printf("l = %d, sb = %d, d = %d, offset = %d,
//sr = %d, over = %d\n", l, sb, d, offset, sr, over);

// Copy increment[...] to leaves of sum[...]

low = 0;
high = size - 1;
spawn(low, high) {
        if($ < under) {
                sum[$ + sbp1] = increment[$]; //1 RTM
        }
        else {
                sum[($ - under) + sb + offset] = increment[$]; //1 RTM
        }
} //1 RTM join

/*
 * LSRTM = 2
 * QD = 0
 * Computation Depth = 5
 * Computation Work = 2N
 */

// Make some 0 leaves at level l+1 so all nodes have exactly
// k children

fill = (k - (under % k)) % k;
for(i = 0; i < fill; i++) {
        sum[sbp1 + under + i] = 0;
}

/*
 * Computation Cost = 2k + 1
 */

// Iteration 1: fill in all nodes at level l
low = sb;
high = sb + offset - 1;
if(high >= low) {
        spawn(low, high) {
                int count;

                sum[$] = 0;
                for(count = 0; count < k; count++) {
                        sum[$] += sum[k * $ + count + 1];
                }
        }
```

```
}

/*
 * We will count the above "Iteration 1" as 1 iteration in
 * the climbing the tree loop below, for simplicity.
 * This gives an upper bound, since the "Iteration 1"
 * section above does slightly less.
 */

// Climb the tree
level = l;
while(level > 0) {
        level --;
        startindex = (pow(k,level) - 1) / (k - 1);
        layersize = pow(k,level);

        low = startindex;
        high = startindex + layersize - 1;
        spawn(low, high) {
                int count;

                /*
                 * All the sum[X] elements are read at once
                 * for the below loop using prefetching.
                 *
                 * RTMs = 1
                 * (prefetch) Computation depth = k
                 */

                sum[$] = 0;
                for(count = 0; count < k; count++) {
                        sum[$] += sum[k * $ + count + 1];
                }

                /*
                 * Computation Depth = 2k + 1
                 */
        }
} // 1 RTM join

/*
 * For the above stage of climbing the tree:
 * LSRTM = 2 * logN
 * Computation Depth = (3k + 9) * logN + 1
 * Computation Work = (3k + 2) * (N - 1) / (k - 1)
 *
 * The (N - 1) / (k - 1) factor of the work is the
 * number of nodes in a k-ary tree of depth logN - 1
 * [there is no work for the leaves at depth logN]
 *
 * Computation Work / min(p, p_i) =
 * ((3k + 2) * (N - min(p, N-1) - 1) / (k - 1)) / p
 * + (3k + 2) * log_k(p)
 *
 * For each level where number of nodes < p, the denominator is p_i.
 * Otherwise the denominator is p.  This gives the above formula.
 */

result[0] = sum[0];

/*
 * For the whole algorithm:
 *
 * LSRTM = 2 * logN + 1
 * QD = 0
 * Computation Depth = (3k + 9) * logN + 2k + 33
 * Computation Work / min(p, p_i) =
 * ((3k + 2)(N - min(p, N-1) - 1) / (k - 1) + 2N) / p
```

```
         * + (3k + 2)log_k(p)
         */
}
```

A.2 Serial Summation

```
/*
 * void sum(...)
 * Function computes a sum
 *
 * Input:
 * increment[] - an array of increment values
 * k - the value of k to use for the k-ary tree
 * size - the size of the increment[] array
 *
 * Output:
 * sum
 *
 */
void sum(int increment[], int *sum, int k, int size) {
        int i;
        *sum = 0;

        for(i = 0; i < size; i++) {
                *sum += increment[i];
        }

        /*
         * LSRTM = 1
         * At first, 1 RTM is needed to read increment. However, later reads
         * to increment are accomplished with prefetch.
         *
         * QD = 0
         * Computation = 2N
         */
}
```

A.3 Synchronous K-ary Prefix Sum

```
/*
 * void kps(...)
 *
 * The function computes prefix sums by using a k-ary tree.
 * k is defined by the parameter k to the function.
 *
 * Input:
 * increment[] - an array of increment values
 * k - the value of k to use for the k-ary tree
 * size - the size of the increment[] array
 *
 * Output:
 * result[] - this array is filled with the prefix sum on the values
 * of the array increment[]
 *
 */
void kps(int increment[], int result[], int k, int size) {
        register int low, high;
        int height = 20; //note: height should be as large as log_k(size)
        //int layersize[height]; // number of nodes in layer i
        int base[height*size]; // base of leaf is its value after PS
                                    // base of internal node is the base of leftmost leaf
        int sum[height*size]; // the value of sum for a node is the sum of the values
                                   // of increment for all its leaves
```

```
int iteration = 0; // determines the current height in the tree

int temp;
int done; //a loop control variable

int l,  //level where leaves start
        sb,     //index where leaves would start if size is a power of 2
        d,      //size - k^l
        offset, //how much to offset due to size not being power of 2
        sr,             //sb + offset
        over,   //number of leaves at level l
        under,  //number of leaves at level l + 1
        sbp1;   //index of one level higher from sb
int fill; //nodes to fill in with 0 to make all nodes have k children

int level, startindex, layersize;

int i;

/*
 * With non-blocking writes 0 RTM is required to initialize
 * the function parameters: k and size
 * 0 RTM is required to initialize local variables such as height
 */

//Special case if size == 1
if(size == 1) { //the check has 0 RTM because size is cached.
        result[0] = 0;
        return;
}

/*
 * 18 lines of code above, means computation cost = 18 up to this point.
 */

//calculate location for leaves in the complete representation
l = log(size) / log(k);

sb = (pow(k,l) - 1) / (k - 1); //this is derived from geometric series
sbp1 = (pow(k,l+1) - 1) / (k - 1);

d = size - pow(k,l);
offset = CEIL(((double) d) / (k - 1));
sr = sb + offset;
over = pow(k,l) - offset;
under = size - over;

/*
 * Computation cost = 8
 */

//printf("l = %d, sb = %d, d = %d, offset = %d,
//sr = %d, over = %d\n", l, sb, d, offset, sr, over);

// Copy increment[...] to leaves of sum[...]

low = 0;
high = size - 1;
spawn(low, high) {
        if($ < under) {
                sum[$ + sbp1] = increment[$]; //1 RTM
        }
        else {
                sum[($ - under) + sb + offset] = increment[$]; //1 RTM
        }
} //1 RTM join

/*
```

```
 * LSRTM = 2
 * QD = 0
 * Computation Depth = 5
 * Computation Work = 2N
 */

// Make some 0 leaves at level l+1 so all nodes have exactly
// k children

fill = (k - (under % k)) % k;
for(i = 0; i < fill; i++) {
        sum[sbp1 + under + i] = 0;
}

/*
 * Computation Cost = 2k + 1
 */

// Iteration 1: fill in all nodes at level l
low = sb;
high = sb + offset - 1;
if(high >= low) {
        spawn(low, high) {
                int count;

                sum[$] = 0;
                for(count = 0; count < k; count++) {
                        sum[$] += sum[k * $ + count + 1];
                }
        }
}

/*
 * We will count the above "Iteration 1" as 1 iteration in
 * the climbing the tree loop below, for simplicity.
 * This gives an upper bound, since the "Iteration 1"
 * section above does slightly less.
 */

// Climb the tree
level = l;
while(level > 0) {
        level --;
        startindex = (pow(k,level) - 1) / (k - 1);
        layersize = pow(k,level);

        low = startindex;
        high = startindex + layersize - 1;
        spawn(low, high) {
                int count;

                /*
                 * All the sum[X] elements are read at once
                 * for the below loop using prefetching.
                 *
                 * RTMs = 1
                 * (prefetch) Computation Depth = k
                 */

                sum[$] = 0;
                for(count = 0; count < k; count++) {
                        sum[$] += sum[k * $ + count + 1];
                }

                /*
                 * Computation Depth of loop = 2k + 1
                 */
        }
```

```
} // 1 RTM join

/*
 * For the above stage of climbing the tree:
 * LSRTM = 2 * logN
 * Computation Depth = (3k + 9) * logN + 1
 * Computation Work = (3k + 2) * (N - 1) / (k - 1)
 *
 * The (N - 1) / (k - 1) factor of the work is the
 * number of nodes in a k-ary tree of depth logN - 1
 * [there is no work for the leaves at depth logN]
 *
 * Computation Work / min(p, p_i) =
 * ((3k + 2) * (N - min(p, N-1) - 1) / (k - 1)) / p
 * + (3k + 2) * log_k(p)
 *
 * For each level where number of nodes < p, the denominator is p_i.
 * Otherwise the denominator is p.  This gives the above formula.
 */

base[0] = 0; //set root base = 0

// Descend the tree
startindex = 0;
while(level < l) {
        layersize = pow(k, level);

        low = startindex;
        high = startindex + layersize - 1;
        spawn(low, high) {
                int count, tempbase;

                tempbase = base[$];

                /*
                 * All the sum[X] elements are read at once
                 * for the below loop using prefetching.
                 *
                 * RTMs = 1
                 * (prefetch) Computation Depth = k
                 */

                for(count = 0; count < k; count++) {
                        base[k*$ + count + 1] = tempbase;
                        tempbase += sum[k*$ + count + 1];
                }

                /*
                 *  Computation Depth = 3k;
                 */

        } //1 RTM join

        startindex += layersize;
        level++;
}

// Iteration h: fill in all nodes at level l+1
low = sb;
high = sb + offset - 1;
if(high >= low) {
        spawn(low, high) {
                int count, tempbase;

                tempbase = base[$];

                for(count = 0; count < k; count++) {
                        base[k*$ + count + 1] = tempbase;
```

```
                                tempbase += sum[k*$ + count + 1];
                        }
                }
        }

        /*
         * For simplicity count "Iteration h" as part of
         * the loop to descend the tree.  This gives
         * an upper bound.
         *
         * For the stage of descending the tree:
         * LSRTM = 2 * logN
         * Computation Depth = (4k + 9) * logN + 2
         * Computation Work = (4k + 2) * (N - 1) / (k - 1)
         *
         * The (N - 1) / (k - 1) factor of the work is the
         * number of nodes in a k-ary tree of depth logN - 1
         * [there is no work for the nodes at depth logN]
         *
         * Computation Work / min(p, p_i) =
         * ((4k + 2) * (N - min(p, N-1) - 1) / (k - 1)) / p
         * + (4k + 2) * log_k p
         *
         * For each level where number of nodes < p, the denominator is p_i.
         * Otherwise the denominator is p.  This gives the above formula.
         */

        //Copy to result matrix
        low = 0;
        high = size - 1;
        spawn(low, high) {
                result[$] = base[sr + $]; //1 RTM
        }

        /*
         * For above code:
         * LSRTM = 1
         * Computation Depth = 4
         * Computation Work = N
         */

        /*
         * For the whole algorithm:
         *
         * LSRTM = 4 * logN + 3
         * QD = 0
         * Computation Depth = (7k + 18) * logN + 2k + 39
         * Computation Work = 3N + (7k + 4) * (N - 1) / (k - 1)
         *
         * Computation Work / min(p, p_i) =
         * (3N + (7k + 4) * (N - min(p, p_i) - 1) / (k - 1)) / p
         * + (7k +4) * log_k p
         */
}
```

A.4 No-Busy-Wait K-ary Prefix-Sum

```
/*
 * void kps(...)
 *
 * The function computes prefix sums by using a k-ary tree.
 * k is defined by the parameter k to the function.
 *
 * Input:
 * increment[] - an array of increment values
 * k - the value of k to use for the k-ary tree
```

```
 * size - the size of the increment[] array
 *
 * Output:
 * result[] - this array is filled with the prefix sum on
 * the values of the array increment[]
 *
 */
void kps(int increment[], int result[], int k, int size) {
        register int low, high;
        int height = 20; //note: height should be as large as log_k(size)
        //int layersize[height]; // number of nodes in layer i
        int base[height*size]; // base of leaf is its value after PS,
                                    // base of internal node is the base of leftmost leaf
        int sum[height*size]; // the value of sum for a node is the sum
                                   // of the values of increment for all its leaves
        int isLeaf[height * size]; // if a leaf: 1; if not a leaf: 0
        int passIndex[height * size]; //array for passing index to child threads

        int iteration = 0; // determines the current height in the tree

        int temp;
        int done; //a loop control variable

        int l,  //level where leaves start
                sb,     //index where leaves would start if size is a power of 2
                d,      //size - k^l
                offset, //how much to offset due to size not being power of 2
                sr,             //sb + offset
                over,   //number of leaves at level l
                under,  //number of leaves at level l + 1
                sbp1;   //index of one level higher from sb
        int fill; //nodes to fill in with 0 to make all nodes have k children

        int level, startindex, layersize;

        int i;

        /*
         * With non-blocking writes 0 RTM is required to initialize
         * the function parameters: k and size
         * 0 RTM is required to initialize local variables such as height
         */

        //Special case if size == 1
        if(size == 1) { //the check has 0 RTM because size is cached.
                result[0] = 0;
                return;
        }

        /*
         * 21 lines of code above, means computation cost = 21 up to this point.
         */

        //calculate location for leaves in the complete representation
        l = log(size) / log(k);

        sb = (pow(k,l) - 1) / (k - 1); //this is derived from geometric series
        sbp1 = (pow(k,l+1) - 1) / (k - 1);

        d = size - pow(k,l);
        offset = CEIL(((double) d) / (k - 1));
        sr = sb + offset;
        over = pow(k,l) - offset;
        under = size - over;

        /*
         * Computation cost = 8
         */
```

```
//printf("l = %d, sb = %d, d = %d, offset = %d, sr = %d, over = %d\n", l, sb, d, offset, sr, over

// Copy increment[...] to leaves of sum[...]

low = 0;
high = size - 1;
spawn(low, high) {
        if($ < under) {
                sum[$ + sbp1] = increment[$]; // 1 RTM
                isLeaf[$ + sbp1] = 1;
        }
        else {
                sum[($ - under) + sb + offset] = increment[$]; //1 RTM
                isLeaf[($ - under) + sb + offset] = 1;
        }
} // 1 RTM join

/*
 * For code above:
 *
 * LSRTM = 2
 * Computation Depth = 6
 * Computation Work = 3N
 */

// Make some 0 leaves at level l+1 so all nodes have exactly
// k children

fill = (k - (under % k)) % k;
for(i = 0; i < fill; i++) {
        sum[sbp1 + under + i] = 0;
}

/*
 * Computation Cost = 2k + 1
 */

//Climb tree

low = sr;
high = sr + size + fill - 1;
spawn(low, high) {
        int gate, count, alive;
        int index= $;
        alive = 1;

        while(alive) {
                index = (index - 1) / k;

                gate = 1;
                psm(gate, &gatekeeper[index]); //1 RTM

                if(gate == k - 1) {
                        /*
                         * Using prefetching, the sum[X] elements
                         * in the following loop are read all at once
                         * LSRTM = 1
                         * (prefetching) Computation Depth = k
                         */

                        sum[index] = 0;
                        for(count = 0; count < k; count++) {
                                sum[index] += sum[k*index + count + 1];
                        }

                        if(index == 0) {
                                alive = 0;
```

```
                        }

                        /*
                         * Computation Depth = 2k + 3;
                         */
                }
                else {
                        alive = 0;
                }
        }
} // 1 RTM join

/*
 * For code above:
 *
 * LSRTM = 2 * logN + 1
 * QD = k * logN
 * Computation Depth = (8 + 2k) * (logN + 1) + 6
 * Computation Work = (8 + 2k) * (N - 1) / (k - 1) + 8N
 *
 * The (N - 1) / (k - 1) factor of the work comes
 * from counting the total nodes in a tree with logN - 1
 * levels.  Each of the leaves at level logN only
 * executes the first 8 lines inside the spawn block
 * (that is, up to the check of the gatekeeper) before
 * most die and only 1 thread per parent continues.  This
 * gives the 8N term.
 *
 * Computation Work / min(p, p_i) =
 * ((8 + 2k)*(N - min(p, N-1) - 1)/(k-1) + 8N) / p
 * + (8 + 2k) * log_k p
 *
 * For each level where number of nodes < p, the denominator is p_i.
 * Otherwise the denominator is p.  This gives the above formula.
 */

base[0] = 0; //set root base = 0

low = 0;
high = 0;
spawn(low, high) {
        int count, tempbase;
        int index = $;
        int newID;

        if($ != 0) {
                index = passIndex[$];
        }

        while(isLeaf[index] == 0) {
                tempbase = base[index];

                /*
                 * The k - 1 calls to sspawn can be executed with
                 * a single kspawn instruction.
                 * The elements sum[X] are read all at once using
                 * prefetching.
                 *
                 * LSRTM = 2
                 * (kspawn and prefetching) Computation Depth = k +1
                 */

                for(count = 0; count < k; count++) {
                        base[k*index + count + 1] = tempbase;
                        tempbase += sum[k*index + count + 1];

                        if(count != 0) {
                                sspawn(newID) {
```

```
                                                passIndex[newID] = k*index + count + 1;
                                        }
                                }
                        }

                        index = k*index + 1;

                        /*
                         * Computation Depth = 6k + 1
                         */
                }
        } //1 RTM join

        /*
         * For code above:
         *
         * LSRTM = 2 * logN + 1
         * Computation Depth = (3 + 6k) * logN + 9
         * Computation Work = (3 + 6k) * (N - 1) / (k - 1) + 6N + 6
         *
         * The (N - 1) / (k - 1) factor of the work comes
         * from counting the total nodes in a tree with logN - 1
         * levels.  Each of the leaves at level logN only
         * executes the first 6 lines inside the spawn block
         * (up to the check of isLeaf) before dying. This
         * gives the 6N term.
         *
         * Computation Work / min(p, p_i) =
         * ((3 + 6k)*(N - min(p, N-1) - 1) / (k-1) + 6N + 6)/p
         * + (3 + 6k) * log_k p
         */

        //Copy to result matrix
        low = 0;
        high = size - 1;
        spawn(low, high) {
                result[$] = base[sr + $]; //1 RTM
        } //1 RTM join

        /*
         * LSRTM = 2
         * Computation Depth = 4
         * Computation Work = N
         */

        /*
         * For the whole algorithm:
         *
         * LSRTM = 4 * logN + 6
         * QD = k * logN
         * Computation Depth = (11 + 8k) * logN + 2k + 55
         * Computation Work = (11 + 8k) * (N - 1) / (k - 1) + 18N + 6
         *
         * Computation Work / min(p, p_i) =
         * ((11 + 8k)*(N - min(p, N-1) - 1) / (k-1) + 18N + 6) / p
         * + (11 + 8k)*log_k p
         */
}
```

A.5 Serial Prefix-Sums

```
/*
 * void kps(...)
 *
 * The function computes prefix sums serially.
 *
```

```
 * Input:
 * increment[] - an array of increment values
 * k - the value of k to use for the k-ary tree (not used)
 * size - the size of the increment[] array
 *
 * Output:
 * result[] - this array is filled with the prefix sum on the values of the array increment[]
 *
 */
void kps(int increment[], int result[], int k, int size) {
        int i;
        int PS = 0;

        for(i = 0; i < size; i++) {
                result[i] = PS;
                PS += increment[i];
        }

        /*
         * LSRTM = 1
         * At first, 1 RTM is needed to read increment.  However, later reads
         * to increment are accomplished with prefetch.
         *
         * QD = 0
         * Computation = 3N
         */
}
```

A.6 Flattened Breadth-First Search

```
/* Flattened BFS implementation
 */
psBaseReg newLevelGR,notDone; // global register for ps()

int * currentLevelSet, * newLevelSet, *tmpSet; // pointers to vertex sets

main() {

  int currentLevel;
  int currentLevelSize;
  register int low,high;
  int i;
  int nIntervals;

  /* variables for the edgeSet filling algorithm */
  int workPerThread;
  int maxDegree,nMax; // hold info about heaviest node

  /* initialize for first level */
  currentLevel = 0;
  currentLevelSize = 1;
  currentLevelSet = temp1;
  newLevelSet = temp2;

  currentLevelSet[0] = START_NODE;
  level[START_NODE]=0;
  gatekeeper[START_NODE]=1; // mark start node visited

  /* All of the above initializations can be done with non-blocking writes.
   * using 0 RTM
   * 7 lines of code above, cost = 9 up to this point
   */

  // 0 RTM, currentLevelSize in cache
  while (currentLevelSize > 0) {  // while we have nodes to explore

    /* clear the markers array so we know which values are uninitialized
```

```
 */
low = 0;
high = NTCU - 1; // 0 RTM, NTCU in cache
spawn(low,high) {
  markers[$] = UNINITIALIZED; // 0 RTM, non-blocking write. UNITIALIZED is a constant
  // the final non-blocking write is overlapped with the RTM of the join
} // 1 RTM for join

/* Total for this spawn block + initialization steps before:
 * RTM Time = 1
 * Computation time = 1
 * Computation work = NTCU, number of TCUs.
 */

 /*********************************************************************
  *  Step 1:
  *  Compute prefix sums of the degrees of vertices in current level set
  *********************************************************************/

/*
 * We use the k-ary tree Prefix_sums function.
 * Changes from "standard" prefix_sums:
 *  - also computes maximum element. this adds to computation time of
 *    upward traversal of k-ary tree
 */

// first get all the degrees in an array
low = 0;
high = currentLevelSize-1;
spawn(low,high) {
  register int LR;
  /* prefetch crtLevelSet[$]
   * this can be overlaped with the ps below,
   * so it takes 0 RTM and 1 computation
   */
  LR = 1;
  ps(LR,GR); // 1 RTM
  degs[GR] = degrees[crtLevelSet[$]];
  // 1 RTM to read degrees[crtLevelSet[$]]. using non-blocking write
  // last write is overlapped with the RTM of the join
} // 1 RTM for join

/* the above spawn block:
 *  RTM Time = 3
 *  Computation Time = 3
 *  Computation Work = 3*Ni
 */

 kary_psums_and_max(degs,prefix_sums,k,currentLevelSize, maxDegree);

/*
 * this function has:
 *  RTM Time = 4 log_k (Ni)
 *  Computation Time = (17 + 9k) log_k(Ni) + 13
 *  Computation Work = (17 + 9k) Ni + 13
 */
outgoingEdgesSize = prefix_sums[currentLevelSize + 1]; // total sum. 0 RTM (cached)

/* compute work per thread and number of edge intervals
 * cost = 3 when problem is large enough, cost = 5 otherwise
 * no RTMs,everything is in cache and using non-blocking writes
 */
nIntervals = NTCU; // constant
workPerThread = outgoingEdgesSize / NTCU + 1;
if (workPerThread < THRESHOLD) {
  workPerThread = THRESHOLD;
  nIntervals = (outgoingEdgesSize / workPerThread) + 1;
}
```

```
/* Total Step 1:
 * RTM Time: 4 log_k Ni + 4
 * Computation Time: (17+9k) log_k Ni + 23
 * Computation Work: (19+9k) Ni + 21
 */

/*********************************************************************
 * Step 2:
 *  Apply parallel pointer jumping algorithm to find all marker edges
 *********************************************************************/

nMax = maxDegree / workPerThread; // 0 RTM, all in cache

/* Step 2.1 Pointer jumping - Fill in one entry per vertex */
low = 0;
// one thread for each node in current layer
high = currentLevelSize - 1;
spawn(low,high) {
  int crtVertex;
  int s,deg;
  int ncrossed;

  /*
   * prefetch currentLevelSet[$], prefix_sums[$]
   * 1 RTM, computation cost = 2
   */

  crtVertex = currentLevelSet[$];  // 0 RTM, value is in cache
  s = prefix_sums[$] / workPerThread + 1; // 0 RTM, values in cache
  // how many (if any) boundaries it crosses.
  ncrossed = (prefix_sums[$] +  degrees[crtVertex]) / workPerThread - s;
    // above line has 1 RTM, degrees[] cannot be prefetched above, depends on crtVertex
  if (ncrossed>0) { // crosses at least one boundary
    markers[s] = s * workPerThread - prefix_sums[$]; // this is the edge index (offset)
    markerNodes[s] = $;  // this is the vertex
  }
  // last write is overlapped with the RTM of the join
} // 1 RTM for join


/*
 * Total for the above spawn block
 *  RTM Time = 3
 *
 * Computation Time = 9
 * Computation Work <= 9 Ni
 */

/* Step  2.2 Actual pointer jumping */

jump = 1; notDone = 1;
while (notDone) { // is updated in parallel mode, 1 RTM to read it
  notDone = 0; // reset
  low=0; high = NTCU-1;
  spawn(low,high) {
    register int LR;
    // will be broadcasted: jump, workPerThread, UNINITIALIZED constant
    /* Prefetch: markers[$], markers[$-jump]
     *  1 RTM, 2 Computation, 1 QD
     */
    if (markers[$] == UNINITIALIZED) { // 0 RTM, cached
      if (markers[$-jump] != UNINITIALIZED) { // 0 RTM, cached
        // found one initialized marker
        markers[$] = markers[$-jump] + s * workPerThread;
        markerNodes[$] = markerNodes[$-jump];
      }
      else { // marker still not initialized. mark notDone
        LR = 1;
```

```
        ps(LR,notDone);   // 1 RTM
      }
    }

  } // 1 RTM for join
  /* Total for the above spawn block + setup
   *  RTM Time = 3
   *  Computation time =  6
   *  Computation work = 6
   *
   */
  jump = jump * 2; // non-blocking write
}

/* above loop executes at most log NTCU times
 *  Total:
 *   RTM Time = 4 log NTCU
 *   Computation time = 10 log NTCU (includes serial code)
 *   Computation work = 6 NTCU
 */


/* Total step 2:
 *  RTM = 4 log NTCU + 3
 *  Computation depth = 10 log NTCU + 9
 *  Computation work. section 1: 9Ni, section 2=10 NTCU
 */

/*****************************************************************
 * Step 3.
 * One thread per edge interval.
 * Do work for each edge, add it to new level if new
 *****************************************************************/

low = 0;
high = nIntervals;      // one thread for each interval
newLevelGR = 0; // empty set of nodes
spawn(low,high) {
  int crtEdge,freshNode,antiParEdge;
  int crtNode,i3;
  int gatekLR; // local register for gatekeeper psm
  int newLevelLR; // local register for new level size

  /*
   * Prefetch markerNodes[$], markers[$]
   *  1 RTM, computation cost 2
   */

  crtNodeIdx = markerNodes[$]; // cached, 0 RTM
  crtEdgeOffset = markers[$]; // cached, 0 RTM

  /* prefetch currentLevelSet[crtNodeIdx],
              vertices[currentLevelSet[crtNodeIdx]],
              degrees[currentLevelSet[crtNodeIdx]]
   * 2 RTM, cost = 2
   */

  // workPerThread is broadcasted, 0 RTM to read it
  for (i3=0;i3<workPerThread;i3++) {
    crtEdge = vertices[currentLevelSet[crtNodeIdx]] + crtEdgeOffset; // cached, 0 RTM
    // traverse edge and get new vertex
    freshNode = edges[crtEdge][1]; // 1 RTM
    if (freshNode!= -1) { // edge could be marked removed
      gatekLR = 1;
      psm(gatekLR,&gatekeeper[freshNode]); // 1 RTM, queuing for the indegree

      if (gatekLR == 0) { // destination vertex unvisited
        newLevelLR = 1;
```

```
        // increase size of new level set
        ps(newLevelLR,newLevelGR); // 1 RTM
        // store fresh node in new level. next two lines are 0 RTM, non-blocking writes
        newLevelSet[newLevelLR] = freshNode;
        level[freshNode] = currentLevel + 1;
        // now mark antiparallel edge as deleted
        antiParEdge = antiParallel[crtEdge]; // 0 RTM, prefetched
        edges[antiParEdge][1] = -1; edges[antiParEdge][0] = -1; // 0 RTM, non-blocking writes

      } // end if
    } // end if freshNode

    /* Previous if block costs:
     * 2 RTM, computation 8 for a "fresh" vertex
     * or
     * 1 RTM, computation 2 for a "visited" vertex
     */

    crtEdgeOffset++;
    if (crtEdgeOffset>=degrees[currentLevelSet[crtNodeIdx]]) { // exhausted all the edges?
      // 0 RTM, value is in cache
      crtNodeIdx++;
      crtEdgeOffset = 0;
      /* We have new current node. prefetch its data
         prefetch currentLevelSet[crtNodeIdx],
       *   vertices[currentLevelSet[crtNodeIdx]],
       *      degrees[currentLevelSet[crtNodeIdx]]
       * 2 RTM, cost = 2
       */
    }

    /* This if and instruction before it cost:
     * 2 RTM, 6 computation for each new marker edge in interval
     * or
     * 2 computation for all other edges
     */

    if (crtNodeIdx>= currentLevelSet)
      break;
    // this if is 0 RTM, 1 computation.

  } // end for

  /* Previous loop is executed C = Ei/p times.
   * We assume Ni nodes are "fresh", worst case analysis
   * Total over all iterations. AA is the number of marker edges in interval.
   *  WITHOUT PREFETCHING:
   *   RTM: 3*C + 2 AA
   *   Computation:  11*C + 4 AA
   */
  // last write is overlapped with the RTM of the join
} // 1 RTM for join

/*
 * Total for above spawn block + initialization: (C=Ei/p, AA = N/p = # marker edges)
 * WITHOUT PREFETCHING for multiple edges: RTM Time = 3*C + 3 + 2 AA
 * WITH PREFETCHING for multiple edges: RTM Time = 3 + 3 + 2
 * Computation Time = 8 + 7*C + 16 AA
 * Computation Work = 8p + 7E + 16N
 */

// move to next layer
currentLevel++;
currentLevelSize = newLevelGR; // from the prefix-sums
// "swap" currentLevelSet with newLevelSet
tmpSet = newLevelSet;
newLevelSet = currentLevelSet;
currentLevelSet = tmpSet;
```

```
    /* all these above steps: 0 RTM, 5 computation */

  } // end while
  /*
   * Total for one BFS level (one iteration of above while loop):
   *   W/O PRE: RTM Time = 4 log_k Ni + 4 |Ei|/p + 11 + LSRTM of PSUMS
   *   W PRE  : RTM Time = 4 log_k Ni + 4  + 11 + LSRTM of PSUMS
   *  Computation Time =
   *  Comp Work =
   */
}
```

A.7 Single-Spawn Breadth-First Search

```
/* BFS implementation using single-spawn operation
 * for nesting
 */
psBaseReg newLevelGR; // global register for new level set

int * currentLevelSet, * newLevelSet, *tmpSet;  // pointers to level sets

main() {

  int currentLevel;
  int currentLevelSize;
  int low,high;
  int i;

  currentLevel = 0;
  currentLevelSize = 1;
  currentLevelSet = temp1;
  newLevelSet = temp2;

  currentLevelSet[0] = START_NODE; // store the vertex# this thread will handle

  /*
   * 0 RTMs, 5 computation
   */

  while (currentLevelSize > 0) {  // while we have nodes to explore
    newLevelGR = 0;
    low = 0;
    high = currentLevelSize - 1; // one thread for each node in current layer

    spawn(low,high) {
      int gatekLR, newLevelLR, newTID;
      int freshNode,antiParEdge;

      /*
       * All threads need to read their initialization data
       *   nForks[$] and currentEdge[$]
       */
      if ($ < currentLevelSize ) { // 0 RTM
        /*
         * "Original" threads read it explicitly from the graph
         */
        // only start degree-1 new threads, current thread will handle one edge
        nForks[$] = degrees[currentLevelSet[$]] - 1; // 2 RTM
        // this thread will handle first outgoing edge
        currentEdge[$] = vertices[currentLevelSet[$]]; // 1 RTM
      }
      else {
        /*
         * Single-spawned threads, need to "wait" until init values
         * from the parent are written
```

```
    */

  while (locks[$]!=1) ;  // busy wait until it gets the signal

} // end if

/* The above if block takes
 *  3 RTM, 3 computation for "original" threads
 *  for child threads: 1 RTM for synchronization. 2 computation
 */

while (nForks[$] > 0) { // 1 computation
  // this is executed for each child thread spawned
   sspawn(newTID) { // 1 RTM
    /*
     * writing initialization data for child threads.
     * children will wait till this data is commited
     */
     nForks[newTID] = (nForks[$]+1)/2 -1;
     nForks[$] = nForks[$] - nForks[newTID]-1;
     currentEdge[newTID] = currentEdge[$] + nForks[$]+1;
     locks[newTID] = 1; // GIVE THE GO SIGNAL!

      /*
       *  0 RTM
       *  4 computation
       */
   }
   /* For each child thread:
    *  1 RTM
    *  5 computation
    */
} // done with forking

/*
 * Prefetch edges[currentEdge[$]][1], antiParallel[currentEdge[$]]
 *  1 RTM, 2 computation
 */

// let's handle one edge
freshNode = edges[currentEdge[$]][1]; // 0 RTM, value was prefetched
if (freshNode != -1 ) { // if edge hasn't been deleted

  gatekLR = 1;
  // test gatekeeper
  psm(gatekLR,&gatekeeper[freshNode]);  // 1 RTM. GQD queuing

  if (gatekLR == 0) { // destination vertex unvisited!
    newLevelLR = 1;
    // increase size of new level set
    ps(newLevelLR,newLevelGR);  // 1 RTM
    // store fresh node in new level
    newLevelSet[newLevelLR] = freshNode;
    level[freshNode] = currentLevel + 1;
    // now mark antiparallel edge as deleted
    antiParEdge = antiParallel[currentEdge[$]]; // 0 RTM, value was prefetched
    edges[antiParEdge][1] = -1;
    edges[antiParEdge][0] = -1;
  } // end if
} // end if

/*
 * Previous if block costs:
 *  2 RTM, 10 computation for "fresh" vertex
 *  0 RTM, 2 computation for visited vertex
   */

  /*
```

```
        * Final write is blocking, but the RTM overlaps the join.
        */
    } // 1 RTM join

    /* Computation for a child thread that starts one single child: 19 */


    // move to next layer
    currentLevel++;
    currentLevelSize = newLevelGR; // from the prefix-sums
    // "swap" currentLevelSet with newLevelSet
    tmpSet = newLevelSet;
    newLevelSet = currentLevelSet;
    currentLevelSet = tmpSet;

    /* the above 5 lines of code: 0 RTM, 5 computation */

  } // end while
}
```

A.8 K-Spawn Breadth-First Search

The only difference between the single-spawn BFS algorithm and the k-spawn is the while loop that is starting children threads. We're including only that section of the code here, the rest is identical with the code in the BFS Single-Spawn implementation.

```
    while (nForks[$] > 0) { // 1 computation
      // this is executed for each child thread spawned
       kspawn(newTID) { // 1 RTM for kSpawn
         // newTID is the lowest of the k TIDs allocated by k-spawn.
         // The other ones are newTID+1, newTID+2,..., newTID+(k-1)
        /*
         * writing initialization data for child threads.
         * children will wait till this data is commited
         */

         slice = nForks[$] / k;
         nForks[$] = nForks[$] - slice; // substract a slice for parent thread

         for (child=0;child<k;child++) {
           // initialize nForks[newTid + child] and currentEdge[newTid + child]
           nForks[newTID + child] = max(slice,nForks[$]); // for rounding
           currentEdge[newTID] = currentEdge[$] + child * slice;
           nForks[$] = nForks[$] - nForks[newTID + child];
           locks[newTID + child] = 1; // GIVE THE GO SIGNAL!
         }
          /*
           * loop is executed k times.
           * Each iterration:
           *  0 RTM
           *  4 computation
           */
       }
       /* For each k child threads:
        *  1 RTM
        *  2+4*k computation
        */
    } // done with forking
```

Bibliography

[ABC+06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[ABD+09] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, 2009.

[ACC+90] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The tera computer system. In *Proceedings of the 4th international conference on Supercomputing*, ICS '90, pages 1–6, New York, NY, USA, 1990. ACM.

[AG92] Farid Alizadeh and Andrew V. Goldberg. Implementing the push-relabel method for the maximum flow problem on a connection machine. Technical Report STAN-CS-92-1410, Department of Computer Science, Stanford University, 1992.

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[AS92] Richard J. Anderson and J. Setubal. On the parallel implementation of goldberg's maximum flow algorithm. In *SPAA '92: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 168–177, New York, NY, USA, 1992. ACM.

[AS93] Richard J. Anderson and J. Setubal. Goldberg's algorithm for maximum flow in perspective: a computational study. *In D. S. Johnson and C. C. McGeoch, editors, Network Flows and Matching: First DIMACS Implementation Challenge*, pages 1–18, 1993.

[BBF+97] P. Bach, M. Braun, A. Formella, J. Friedrich, T. Grun, and C. Lichtenau. Building the 4 processor sb-pram prototype. In *System Sciences, 1997, Proceedings of the Thirtieth Hawaii International Conference on*, volume 5, pages 14 –23 vol.5, jan 1997.

[BBH+04] Darrell Boggs, Aravindh Baktha, Jason Hawkins, Deborah T. Marr, J. Alan Miller, Patrice Roussel, Ronak Singhal, Bret Toll, and K.S. Venkatraman. The microarchitecture of the intel pentium 4 processor on 90nm technology. *Intel Technology Journal*, 8(1):7–23, February 2004.

[BCF05] David A. Bader, Guojing Cong, and John Feo. On the architectural requirements for efficient execution of graph algorithms. In *Proceedings of the 2005 International Conference on Parallel Processing*, pages 547–556, Washington, DC, USA, 2005. IEEE Computer Society.

[BG09] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 2009. ACM.

[BHQV07] Aydin O. Balkan, Michael N. Horak, Gang Qu, and Uzi Vishkin. Layout-accurate design and implementation of a high-throughput interconnection network for single-chip parallel processing. *hoti*, pages 21–28, 2007.

[Bre74] R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, 1974.

[BS05] David Bader and Vipin Sachdeva. A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. In *PDCS '05: Proceedings of the 18th ISCA International Conference on Parallel and Distributed Computing Systems*, 2005.

[BV06] A. O. Balkan and U. Vishkin. Programmer's manual for xmtc language, xmtc compiler and xmt simulator. Technical Report UMIACS-TR 2005-45, University of Maryland Institute for Advanced Computer Studies (UMIACS), February 2006.

[CB95] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *Computers, IEEE Transactions on*, 44(5):609–623, May 1995.

[CBM+09] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, October 2009.

[CG97] B. V. Cherkassky and A. V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19:390–410, 1997. 10.1007/PL00009180.

[CGS97] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1997.

[Che79] B. V. Cherkassky. *Collected Papers, Issue 3: Combinatorial Methods for Flow Problems*, chapter A Fast Algorithm for Computing MAximum Flow in a Network, pages 90–96. Institute for System Studies, Moscow, 1979. In Russian. English translation appears in AMS Translations, Vol. 158, pp. 23-30. AMS, Providence, RI, 1994.

[CKP+93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: towards a realistic model of parallel computation. *SIGPLAN Not.*, 28:1–12, July 1993.

[CKTV10] George C. Caragea, Fuat Keceli, Alexandros Tzannes, and Uzi Vishkin. General-purpose vs. gpu: Comparison of many-cores on irregular workloads. In *HotPar '10: Proceedings of the 2nd Workshop on Hot Topics in Parallelism*. USENIX, June 2010.

[CM98] J. Cheriyan and K. Mehlhorn. An analysis of the highest-level selection rule in the preflow-push max-flow algorithm. *Information Processing Letters*, 69:69–239, 1998.

[CMCH91] William Y. Chen, Scott A. Mahlke, Pohua P. Chang, and Wen Mei W. Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *MICRO 24: Proceedings of the 24th annual international symposium on Microarchitecture*, pages 69–73, New York, NY, USA, 1991. ACM Press.

[CSWV09] George C. Caragea, A. Beliz Saybasili, Xingzhi Wen, and Uzi Vishkin. Brief announcement: performance potential of an easy-to-program pram-on-chip prototype versus state-of-the-art processor. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 163–165, New York, NY, USA, 2009. ACM.

[CT65] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[CTK+10] George C. Caragea, Alexandros Tzannes, Fuat Keceli, Rajeev Barua, and Uzi Vishkin. Resource-aware compiler prefetching for many-cores. In *Proc. International Symposium on Parallel and Distributed Computing (ISPDC)*, Istanbul, Turkey, July 2010. IEEE Computer Society.

[CTK+11] George C. Caragea, Alexandros Tzannes, Fuat Keceli, Rajeev Barua, and Uzi Vishkin. Resource-aware compiler prefetching for fine-grained many-cores. *International Journal of Parallel Programming (IJPP)*, 2011.

[CV86] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 206–219, New York, NY, USA, 1986. ACM Press.

[CV11] George C. Caragea and Uzi Vishkin. Brief announcement: Better speed-ups for parallel max-flow. In *In proc. ACM Symposium of Parallel Algorithms and Architectures (SPAA)*, 2011.

[DDS95] Fredrik Dahlgren, Michel Dubois, and Per Stenström. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 6(7):733–746, 1995.

[Din70] E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math. Dokl*, 11:1277–1280, 1970.

[DM89] U. Derigs and W. Meier. Implementing goldberg's max-flow-algorithm - a computational investigation. *Mathematical Methods of Operations Research*, 33(6):383–403, November 1989.

[EG88] D. Eppstein and Z. Galil. Parallel algorithmic techniques for combinatorial computation. *Annual review of computer science: vol. 3, 1988*, pages 233–283, 1988.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19:248–264, April 1972.

[FF62] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, New Jersey, 1962.

[FHKK05] John Feo, David Harper, Simon Kahan, and Petr Konecny. Eldorado. In *Proceedings of the 2nd conference on Computing frontiers*, CF '05, pages 28–34, New York, NY, USA, 2005. ACM.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[Gar85] Harold N. Garbow. Scaling algorithms for network problems. *Journal of Computer and System Sciences*, 31(2):148 – 168, 1985.

[GG88] Donald Goldfarb and Michael Grigoriadis. A computational comparison of the dinic and network simplex methods for maximum flow. *Annals of Operations Research*, 13:81–123, 1988. 10.1007/BF02288321.

[GGH91] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. *SIGPLAN Not.*, 26(4):245–257, 1991.

[GGK+82] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The nyu ultracomputer: designing a mimd, shared-memory parallel machine (extended abstract). In *ISCA '82: Proceedings of the 9th annual symposium on Computer Architecture*, pages 27–42, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[GH98] E.M. Gagnon and L.J. Hendren. Sablecc, an object-oriented compiler framework. In *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings*, pages 140 –154, August 1998.

[GMR98] P. B. Gibbons, Y. Matias, and V. Ramachandran. The queue-read queue-write asynchronous pram model. *Theor. Comput. Sci.*, 196:3–29, April 1998.

[Gol87] Andrew V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers*. PhD thesis, M.I.T., 1987.

[Gol98] Andrew V. Goldberg. Recent developments in maximum flow algorithms (invited lecture). In *SWAT '98: Proceedings of the 6th Scandinavian Workshop on Algorithm Theory*, pages 1–10, London, UK, 1998. Springer-Verlag.

[Gol06] Andrew Goldberg. Network optimization library. http://www.avglab.com/andrew/soft.html, 2006.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.

[GT89] Andrew Goldberg and Robert Tarjan. A parallel algorithm for finding a blocking flow in an acyclic network. *Information Processing Letters*, 31:265–271, 1989.

[GT90] Andrew V. Goldberg and Robert E. Tarjan. Finding minimum-cost circulations by successive approximation. *Math. Oper. Res.*, 15:430–466, July 1990.

[GV06] P. Gu and U. Vishkin. Case study of gate-level logic simulation on an extermely fine-grained chip multiprocessor. *Journal of Embedded Computing, Special Issue on Embedded Single-Chip Multicore Architectures*, 2:181–190, 2006.

[GVH01] T. Givargis, F. Vahid, and J. Henkel. System-level exploration for pareto-optimal configurations in parameterized systems-on-a-chip. *Computer Aided Design, 2001. ICCAD 2001. IEEE/ACM International Conference on*, pages 25–30, 2001.

[HB09] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2009. Version 1.1.

[HBK01] Jaehyuk Huh, Doug Burger, and Stephen W. Keckler. Exploring the design space of future cmps. *Parallel Architectures and Compilation Techniques, International Conference on*, 0:0199, 2001.

[HH10] Zhengyu He and Bo Hong. Dynamically tuned push-relabel algorithm for the maximum flow problem on cpu-gpu-hybrid platforms. In *The 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS'10)*, 2010.

[HJ99] David R. Helman and Joseph JáJá. Designing practical efficient algorithms for symmetric multiprocessors. In *Selected papers from the International Workshop on Algorithm Engineering and Experimentation*, ALENEX '99, pages 37–56, London, UK, 1999. Springer-Verlag.

[HN07] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *In proceedings High Performance Computing - HiPC*, pages 197–208, 2007.

[HNR90] P. Heidelberger, A. Norton, and J.T. Robinson. Parallel quicksort using fetch-and-add. *IEEE Transactions on Computers*, 39(1):133–138, Jan 1990.

[HVD07] Mohamed Hussein, Amitabh Varshney, and Larry Davis. On implementing graph cuts on cuda. In *First Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, 2007.

[JáJ92] Joseph JáJá. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[JM93] D. S. Johnson and C. C. McGeoch, editors. *Network Flows and Matching: First DIMACS Implementation Challenge*. AMS, Providence, RI, 1993.

[JN09] Magnus Jahre and Lasse Natvig. A high performance adaptive miss handling architecture for chip multiprocessors. *Transactions on High-Performance Embedded Architectures and Compilers*, 4(1), 2009.

[Jou90] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 364–373, New York, NY, USA, 1990. ACM Press.

[Kar74] A. V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Math. Dokl*, 15, 1974.

[KKT01] J. Keller, C. Keßler, and J. Träff. *Practical PRAM programming*. John Wiley & Sons, 2001.

[KL91] Alexander C. Klaiber and Henry M. Levy. An architecture for software-controlled data prefetching. In *ISCA '91: Proceedings of the 18th annual international symposium on Computer architecture*, pages 43–53, New York, NY, USA, 1991. ACM Press.

[KR90] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. *Handbook of theoretical computer science (vol. A): algorithms and complexity*, pages 869–941, 1990.

[KTC+11] Fuat Keceli, Alexandros Tzannes, George C. Caragea, Rajeev Barua, and Uzi Vishkin. Toolchain for programming, simulating and studying the xmt many-core architecture. In *Proc. International Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2011.

[LF80] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *J. ACM*, 27:831–838, October 1980.

[LLB+06] Yingmin Li, B. Lee, D. Brooks, Zhigang Hu, and K. Skadron. Cmp design space exploration subject to physical constraints. *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 17–28, Feb. 2006.

[LNOM08] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[LRB01] Wi Fen Lin, Steven K. Reinhardt, and Doug Burger. Reducing dram latencies with an integrated memory hierarchy design. *hpca*, 00:0301, 2001.

[LvdWDV99] P. Lieverse, P. van der Wolf, E. Deprettere, and K. Vissers. A methodology for architecture exploration of heterogeneous signal processing systems. *Signal Processing Systems, 1999. SiPS 99. 1999 IEEE Workshop on*, pages 181–190, 1999.

[LWmH10] Lijuan Luo, Martin Wong, and Wen mei Hwu. An effective gpu implementation of breadth-first search. In *Proc. of Design Automation Conference (DAC)*, 2010. To appear.

[McI98] Nathaniel McIntosh. *Compiler support for software prefetching*. PhD thesis, Rice University, May 1998. Adviser-Ken Kennedy.

[MLG92] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. *SIGPLAN Not.*, 27(9):62–73, 1992.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, page 4, 1965.

[Mow95] Todd Carl Mowry. *Tolerating latency through software-controlled data prefetching*. PhD thesis, Stanford, CA, USA, 1995.

[Mow98] Todd C. Mowry. Tolerating latency in multiprocessors through compiler-inserted prefetching. *ACM Trans. Comput. Syst.*, 16(1):55–92, 1998.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391, New York, NY, USA, 2005. ACM Press.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, 2002. Springer-Verlag.

[NNTV01] Dorit Naishlos, Joseph Nuzman, Chau-Wen Tseng, and Uzi Vishkin. Towards a first vertical prototyping of an extremely fine-grained parallel programming approach. In *SPAA '01: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 93–102, New York, NY, USA, 2001. ACM.

[Nov03] Diego Novillo. Tree ssa: A new optimization infrastructure for gcc. In *Proceedings of the 2003 GCC Developers' Summit*, pages 181–193, 2003.

[NP11] Donald Nguyen and Keshav Pingali. Synthesizing concurrent schedulers for irregular algorithms. In *ASPLOS '11: Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[NVI09] NVIDIA. *NVIDIA CUDA SDK 2.3*. NVIDIA Corporation, Santa Clara, California, 2009.

[NVI10] NVIDIA. Cuda zone. http://www.nvidia.com/cuda, 2010.

[Ope08] Openmp application program interface, ver. 3.0. http://www.openmp.org, May 2008.

[PBB+02] Wolfgang J. Paul, Peter Bach, Michael Bosch, Jörg Fischer, Cédric Lichtenau, and Jochen Röhrig. Real pram programming. In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, Euro-Par '02, pages 522–531, London, UK, 2002. Springer-Verlag.

[PFML09] Allan Porterfield, Rob Fowler, Anirban Mandel, and Min Yeol Lim. Empirical evaluation of multi-socket, multi-core memory concurrency. Technical Report RENCI TR-09-01, Renaissance Computing Institute, January 2009.

[QLMP06] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A case for mlp-aware cache replacement. In *ISCA '06: Proceedings of the*

33rd annual international symposium on Computer Architecture, pages 167–178, Washington, DC, USA, 2006. IEEE Computer Society.

[Rei07] James Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[SHG09] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *Proc. 23rd IEEE International Parallel and Distributed Processing Symposium*, May 2009.

[Ski97] S. Skiena. *The Algorithm Design Manual*. Springer, Nov 1997.

[Smi82] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982.

[Sni08] Mark Snir. Multi-core and parallel programming: Is the sky falling? The Computing Community Consortium Blog, http://www.cccblog.org/2008/11/17/multi-core-and-parallel-programming-is-the-sky-falling/, November 2008.

[STT10] S. Solomon, P. Thulasiraman, and R.K. Thulasiram. Exploiting parallelism in iterative irregular maxflow computations on gpu accelerators. In *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*, pages 297 –304, 2010.

[SV82] Yossi Shiloach and Uzi Vishkin. An o(n2 log n) parallel max-flow algorithm. *J. Algorithms*, 3(2):128–146, 1982.

[TCBV10] Alexandros Tzannes, George C. Caragea, Rajeev Barua, and Uzi Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, pages 179–190, New York, NY, USA, 2010. ACM.

[TCT06] James Tuck, Luis Ceze, and Josep Torrellas. Scalable cache miss handling for high memory-level parallelism. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 409–422, Washington, DC, USA, 2006. IEEE Computer Society.

[TE95] Dean M. Tullsen and Susan J. Eggers. Effective cache prefetching on bus-based multiprocessors. *ACM Trans. Comput. Syst.*, 13(1):57–88, 1995.

[VCL07] Uzi Vishkin, George C. Caragea, and Bryant C. Lee. *Handbook of Parallel Computing: Models, Algorithms and Applications*, chapter Models for Advancing PRAM and Other Algorithms into Parallel Programs for a PRAM-On-Chip Platform. CRC Press, 2007.

[VDBN98] Uzi Vishkin, Shlomit Dascal, Efraim Berkovich, and Joseph Nuzman. Explicit multi-threading (xmt) bridging models for instruction parallelism (extended abstract). In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 140–151, New York, NY, USA, 1998. ACM.

[Vis92] Uzi Vishkin. A parallel blocking flow algorithm for acyclic networks. *J. Algorithms*, 13(3):489–501, 1992.

[Vis97] U. Vishkin. From algorithm parallelism to instruction-level parallelism: an encode-decode chain using prefix-sum. In *SPAA '97: Proceedings of the 9th annual ACM symposium on Parallel algorithms and architectures*, pages 260–271, New York, NY, USA, 1997. ACM Press.

[Vis00] Uzi Vishkin. A no-busy-wait balanced tree parallel algorithmic paradigm. In *SPAA '00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 147–155, New York, NY, USA, 2000. ACM.

[Vis07] Uzi Vishkin. Thinking in parallel: Some basic data-parallel algorithms and techniques. In use as class notes since 1993. `http://www.umiacs.umd.edu/users/vishkin/PUBLICATIONS/classnotes.pd` September 2007.

[VN08] V. Vineet and P.J. Narayanan. Cuda cuts: Fast graph cuts on the gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, pages 1 –8, june 2008.

[Wen08] Xingzhi Wen. *Hardware Design, Prototyping and Studies of the Explicit Multi-Threading (XMT) Paradigm*. PhD thesis, University of Maryland, August 2008.

[WV07] Xingzhi Wen and Uzi Vishkin. Pram-on-chip: first commitment to silicon. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 301–302, New York, NY, USA, 2007. ACM Press.

[WV08a] Xingzhi Wen and Uzi Vishkin. Fpga-based prototype of a pram-on-chip processor. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 55–66, New York, NY, USA, 2008. ACM.

[WV08b] Xingzhi Wen and Uzi Vishkin. The xmt fpga prototype/cycle-accurate-simulator hybrid. In *WARP08: The 3rd Workshop on Architectural Research Prototyping*, Beijing, China, June 2008. In conjunction with ISCA 2008.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.

[XMT10] Software release of the explicit multi-threading (xmt) programming environment. `http://www.umiacs.umd.edu/users/vishkin/XMT/sw-release.html` October 2010. v0.82.

[YYX05] Canqun Yang, Xuejun Yang, and Jingling Xue. *Advances in Computer Systems Archiecture*, volume 3740/2005, chapter Improving the Performance of GCC by Exploiting IA-64 Architectural Features, pages 236–251. Springer-Verlag, 2005.