Università degli Studi di Pisa
Scuola Normale Superiore di Pisa
University of California at San Diego, UCSD

Compiling Issues for the Simultaneous Multithreading Processor

Advisor
prof. Dean M. Tullsen
UCSD

Second advisor
prof. Marco Vanneschi
Univ. di Pisa

Candidate
Diego Puppin

Academic Year
1999/2000

Master's Thesis:
Compiling Issues for the Simultaneous Multithreading Processor

Advisor
prof. Dean M. Tullsen

Second advisor
prof. Marco Vanneschi

Candidate
Diego Puppin
diego.puppin@sns.it

Academic Year 1999/2000

Pisa, July 21, 2000

Compiling Issues for the Simultaneous Multithreading Processor

Abstract

In the run for performance, many new architectures have been proposed as possible successors of present-day commodity processors. In order to achieve higher performance, the idea behind the simultaneous multithreading (SMT) processor is to introduce few changes into an advanced superscalar processor, yet enable it to execute instructions from different threads on the different functional units in the same cycle: thread-level parallelism becomes so a suitable source of instruction-level parallelism.

This work presents experimental results on performance of Livermore loops on the SMT: advanced compiling techniques are discussed and evaluated, that could be implemented into an advanced compiler for the SMT. The general effectiveness of interleaving, loop fusion and some other techniques poses encouraging results in the direction of an advanced multithreading compiler.

A performance model for SMT is also presented: the methodology described here uses compile-time information to determine upper and lower bounds for the parallelized performance of simple loops. The model is flexible enough to manage many different types of kernels, and is tested with the Livermore loops. Good results have been achieved with short kernels compiled with the *GNU C compiler*: with small loops (less than 40 instructions), the actual performance (useful processor utilization) is not farther than 5% from the maximum expected value.

Please report any comments and suggestions to the author.

Elwood: *It's a 106 miles to Chicago. We got a full tank of gas, half a pack of cigarettes, it's dark and we're wearing sunglasses.*

Jake: *Hit it!*

The Blues Brothers (1980)

Acknowledgements

At the end of an extraordinary time of my life, as the one I spent in Pisa, I would like here to express my thanks to all the people that, in different ways, were important in making it so.

First of all, I would like to thank my advisors: professor Marco Vanneschi and professor Dean Tullsen. The former, for his valuable help and trust, during my academic work; the latter, for all the time he placed at my disposal to supervise my research in San Diego.

I owe sincere thanks to the people that shared with me some amazing experiences throughout these years, whose friendship made all this much easier. The Politburo, who made me grow up a lot: our project will go on, even with an ocean in the middle. My mates at Scuola Normale. My mates at the University — the Collettivo Autonomo Informatici and the Amici Birilli (thank you, Matteo) — with whom I will be forever linked by the memory of unique moments. Particular thanks to Luciano, for his inexhaustible warmth and cheer: we have been a wonderful team. Sincere thanks also to Natalina the fairy, for her affection and kindness.

I owe a particular debt to my family, always close to me, who unceasingly showed their great trust in me: I owe them the opportunity I had to spend a peaceful time here in Pisa. I will soon be far from you, but I know you will always be there for me: really thank you so much, this is for you. Thank-you especially to my little brother, whose esteem has always been important to me: I wish you all the best.

These last lines are to Francesca. I would like to thank you for all that you gave me during the two wonderful years we spent together. You are the most beautiful thing that ever happened to me, a continual spur to give the best and a peaceful spot in the difficult moments. Without you, everything would have been different. I will never forget you.

Ringraziamenti

Alla fine di un periodo straordinario della mia vita, come è stato quello trascorso a Pisa, non posso fare a meno di esprimere un dovuto ringraziamento alle persone che, in diversi modi, hanno contribuito a renderlo tale.

Voglio per primi ringraziare i miei relatori: il professor Marco Vanneschi ed il professor Dean Tullsen. Il primo per la preziosa collaborazione e fiducia durante tutto il mio percorso universitario; il secondo per la grande disponibilità con cui ha seguito il mio lavoro a San Diego.

Vorrei inoltre ringraziare le persone con cui ho condiviso bellissime esperienze in questi anni, senza le quali tutto questo sarebbe stato difficile. Il Politburo, la cui amicizia mi ha fatto molto crescere: il nostro progetto andrà avanti, anche con un oceano di mezzo. I miei compagni alla Scuola Normale. I miei compagni di Università — il Collettivo Autonomo Informatici e gli Amici Birilli (un forte abbraccio a Matteo) — a cui sarò sempre unito dal ricordo di momenti irripetibili. Un particolare ringraziamento a Luciano, per il suo inesaribile calore ed entusiasmo: siamo stati una squadra fantastica! Un sincero grazie poi alla fata Natalina, per il suo affetto e la sua semplicità.

Devo poi moltissimo alla mia famiglia, che mi è sempre stata vicina, e che ha sempre versato su di me una grande fiducia: a loro devo la possibilità di aver passato in maniera serena questi anni. Anche se presto sarò lontano, saprò di avervi sempre accanto: grazie davvero, questo è per voi. Grazie in particolare al mio fratellino, la cui stima mi è sempre stata molto importante: ti auguro tutto il meglio.

Queste ultime righe sono per Francesca. Voglio ringraziarti per tutto quello che mi hai dato nei due bellissimi anni passati insieme. Sei la cosa più bella che mi sia mai successa, un continuo stimolo a dare il meglio di me e un'oasi serena nei momenti di stanchezza. Senza di te sarebbe stato tutto molto diverso. Non ti scorderò mai.

Contents

List of Figures

List of Tables

List of Program Fragments

Chapter 1

Introduction

As processors are growing larger and larger (the one-hundred-million-transistor chip is not so far), computer designers are planning directions for next-generation processors. Just scaling present day architecture to larger configurations seems not to guarantee the expected performance, so new ideas are proposed and explored, in order to take full advantage of future engineering opportunities.

Processing-In-Memory (PIM) (BK97), Intelligent RAM (IRAM) (PAC$^+$97), reconfigurable architectures (DeH96, DeH00b), asynchronous VLSI (MLM$^+$97), Raw Architecture Workstation (RAW) (WTS$^+$97) are just some of many proposals. Different problems are faced by these architectures: better memory interface, advanced usage of logic arrays, better usage of intra-chip connections. They offer quite new and original architectural models, in the run for performance.

The main idea behind simultaneous multithreading (SMT[1]) computing, instead, is to create a more efficient and versatile processor, able to *exploit more parallelism, in all its available forms*: such a processor would be able to take advantage of both instruction-level parallelism (ILP) and thread-level parallelism (TLP) with the same ease. A SMT processor will be implemented as a superscalar multiprocessor (with multiple instruction issues) offering multithreaded execution capabilities.

Simultaneous multithreading can be thought as a technique whose main goal is to achieve higher utilization of the computational capabilities of wide superscalar processors. On a SMT pro-

[1]Throughout this thesis, SMT will mean, according to the context, the architectural model, a processor featuring the described capabilities, or the specific processor simulated by means of SMTSIM.

cessor, TLP can come from multithreaded parallel programs or from individual programs in a multiprogrammed work-load.

SMT will use the same instruction-set architecture (ISA) of superscalar processors, and most of their design. This can be a strong point in determining a smooth introduction of SMT features in commodity processors. With the rapid growth of a multithreaded programming style, that is gaining popularity among developers with *Java*, SMT sets itself as a natural candidate to replace present-day superscalar processor, strong of its advanced multithreading capabilities. As a matter of fact, *Compaq* is planning to introduce SMT features into its commodity processors: *Alpha 21464*, to appear in 2003, will be a SMT superscalar processor (Die99).

1.1 The performance challenge

In order to enhance present-day superscalar processors' performance, many solutions have been proposed.

One of these is to use the chip real estate to build larger and larger on-chip memories, as featured by some recent processors (see figure 1.1 (Ald00)). Even if very popular, many studies show that this solution is not enough to gain proportional performance, and beyond a certain point larger caches seem to be not so useful (see for instance (PS96)). Some paradoxical choices further highlight this solution's limit: in order to have a low interprocessor communication latency in *Cray T3D/T3E*, designers removed the second-level cache from the featured *Alpha* processor.

Another one is to increase peak bandwidth, through increasing clock-frequency (with deeper pipelines) and the number of functional units of superscalar processors (see table 1.1) (SFK97, Tul96b). Leaving all the engineering problems related to this design (huge monolithic core) out of account for the moment, it should be remembered that compilers' ability to extract ILP is still limited, as well as the opportunities that run-time structures (reordering, renaming...) have to remove dependencies. John Hennessy, in his speech at Microprocessor Forum 1999, showed that 4-wide superscalar processors rarely achieve a sustained a 2 instruction-per-cycle rate under *Spec95* benchmarks (Die99).

Even with advanced features as out-of-order execution and register renaming, performance is affected by instruction dependen-

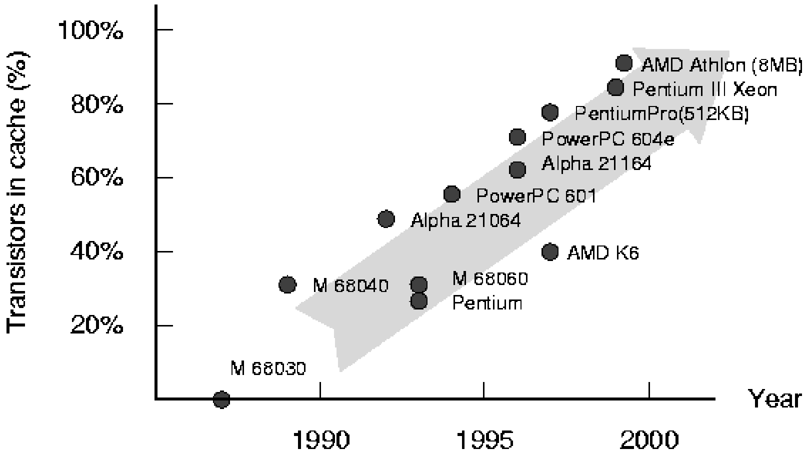


Figure 1.1: Dimension of cache in some recent processors

Architecture (year)	Issue bw.
Power1 (1990)	4
Power2 (1993)	6
PowerPC 601 (1993)	3
PowerPC 603 (1993)	3
PowerPC 604 (1995)	4
PowerPC 620 (1996)	4
α21064 (1992)	2
α21064A (1993)	2
α21164 (1995)	4
HP PA7100 (1992)	2
HP PA7200 (1995)	2
HP PA8000 (1996)	4
Pentium Pro (1995)	3
MIPS R10000 (1994)	5

Table 1.1: Evolution of issue bandwidth in some modern superscalar microprocessors

cies, that limit instruction issuing. Processor can suffer from so-called *vertical wastes*, when all the functional units are idle for one or more cycles, due to data dependencies.

Fine-grain multithreaded processors, able to change context every cicle without degradation, seem to be a good approach to this problem, as they are able to interleave the execution of different threads in order to hide dependencies and latencies. Some studies anyway hold that they are not able to utilize more than 40% of a wide superscalar execution bandwidth (TEL95). It should be clear, however, that even if vertical wastes are avoided by multithreading, a single thread may not be able to fill the whole execution bandwidth, due to the limited ILP it can offer. This problem is known as *horizontal waste*, and only a more efficient exploiting of ILP can face this problem.

A strong debate is going on between ILP pessimists and optimists. ILP advocates hold that ILP is abudant, and can be exploited with a few more ten millions of transistors, and a little compiler magic (see for instance Itanium (MPR99a) and Sparc 64 V (MPR99b) projects). Yale Patt at University of Texas and John Shen at Carnegie-Mellon University believe that advanced superscalar techniques, such as static scheduling, prediction, trace-processing and superspeculation, will allow to ILP to scale suitably: in their opinion a 16- or 32-wide superscalar processor will sustain a ILP of more than 10 instructions per cycle (Die99).

These processors will feature a very large monolithic core, along with the related complexity of design and testing. Explicit thread-level parallelism (TLP) can be seen as a way to keep processors simpler. Chip multiprocessors (CMP), as IBM Power4 (MPR99c) and Sun MAJC (MPR99d) trust in the parallelism that can be found between different threads. These projects will implement, on a single chip, a few replicated superscalar processors: there will be the opportunity to run multiple threads, each one on an independent complete advanced processor (with smaller execution bandwidth).

Nonetheless, both these kinds of architecture (ILP- or TLP-oriented) suffer from poor utilization if the work-load does not match the design parameters: they shows no flexibility when the parallelism moves from ILP to TLP or *vice versa*.

1.2 SMT: TLP as useful ILP

This distinction between instruction-level and thread-level parallelism holds no more with SMT: in this new architectural model, they both represent a way to find independent instructions, that can be executed in parallel. On a SMT, instructions coming from different threads compete for the shared processor resources every cycle. SMT is able to transform the parallelism present among instruction from different threads into instruction-level parallelism, and exploit the whole execution bandwidth computing for different threads in the same cycle. Simultaneous multithreading is a new way to take full advantage of parallelism in all its form, being able to get full utilization with both TLP and ILP, and to adapt to their dynamic changes, without any degradation: when only one thread is available, it can exploit all the functional units, as in one traditional superscalar processor, but when ILP is low, more threads can run and fill the execution bandwidth with their instructions.

Figure 1.2 gives the reader more insight about the main differences between the discussed architectures. The reader is adived to focus on what is referred to as *horizontal* and *vertical waste*, a formal definition of which should be now clearer:

- *vertical waste*: with this term, I mean, following (Tul96b), a clock-cycle in which all the functional units are unused; this is due to instruction dependences, and can be observed in superscalar processor and multiprocessors-on-chip (see figure 1.2.a and 1.2.d);
- *horizontal waste*: this term refers to unused functional units during a clock-cycle; we have this type of degradation when the available ILP is not enough to fill all the execution bandwidht; this problem is common to all the discussed architectures, even if SMT tries to reduce this effect, using TLP as ILP.

In (BG00) the related terms *horizontal* and *vertical sharing* are introduced:

- *vertical sharing*: it is the capability of switching among different threads in different clock cycle (fine-grain multithreading);
- *horizontal sharing*: the unique capability of SMT to share functional units, allowing instructions from different threads to be

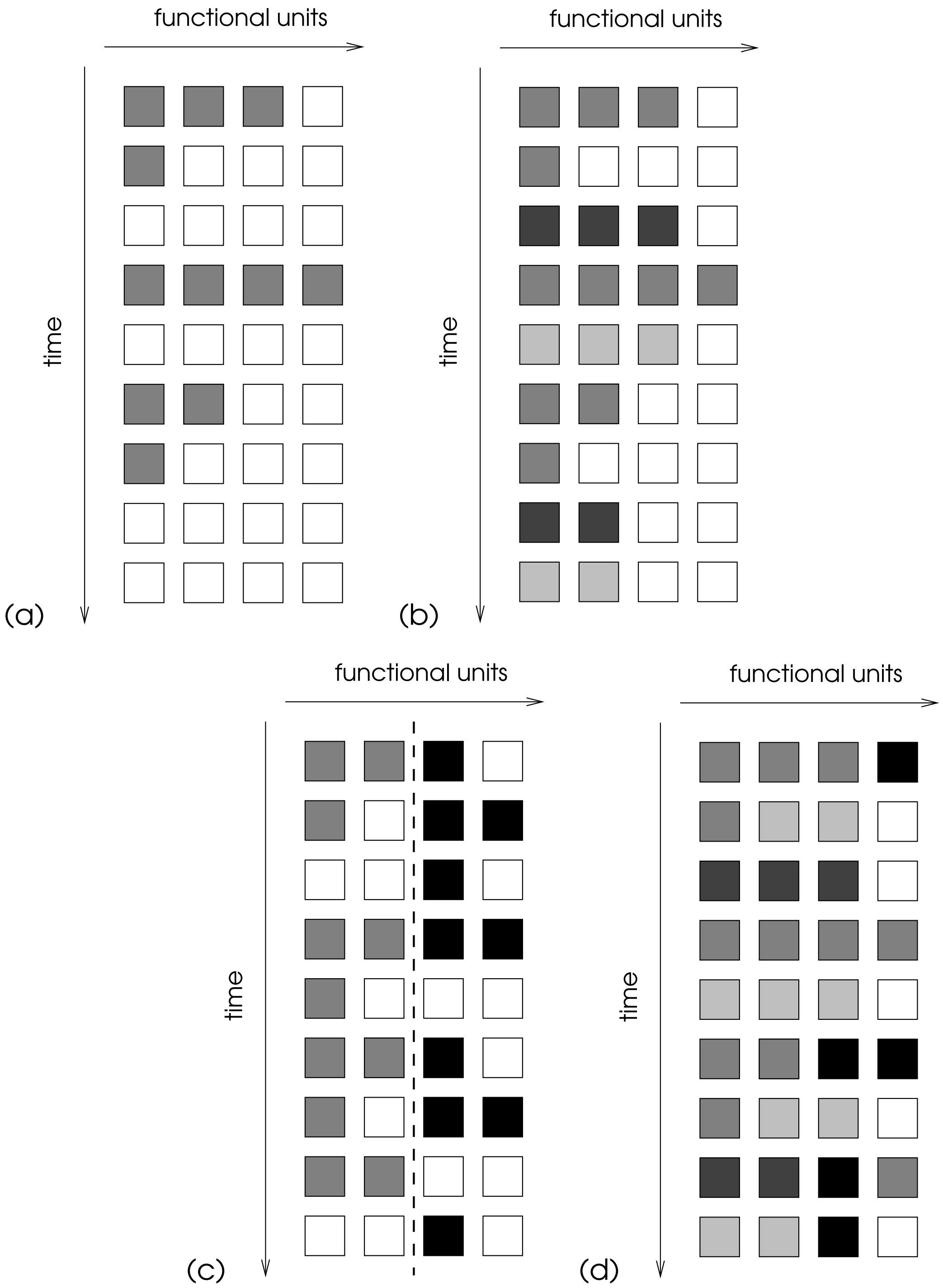


Figure 1.2: ILP and TLP in different architectures: superscalar (a), fine-grained MT (b), dual on-chip processor (c), SMT (d). Different shades correspond to instructions from different threads, unused slots are shown in white.

executed in the same cycle, exploiting the parallelism present among different functional units.

The term *slot* will also be used, referring to execution bandwidth: a superscalar processor, with 8 fully pipelined functional units, offers 8 slots each cycle. Instructions compete for free slots.

1.2.1 SMT architecture

SMT architecture, as presented in (LEE^{+}97), is based on *MIPS R10000* (Yea96), an advanced out-of-order superscalar processor. As said, SMT is able to execute, in a single cycle, instructions from different threads. This unique capability is implemented with very limited changes to the pipeline (see figure 1.3), the most important of which are replication of instruction counters (one per thread) and introduction of a smart trick in the register renaming phase of the pipeline, as described below.

Fetching mechanism is modified to make SMT able to fetch four instructions from two threads every cycle, instead of eight instructions from a single thread as in the original architecture. The total fetching bandwidth is not modified, and only two ports to the instruction cache are required. Fetched threads are chosen among the ones not incurring in a cache miss, with a priority policy called *icount* (TEE^{+}96). This technique assigns highest priority to threads having the least number of instructions in the decode, renaming, and queue pipeline stages. This favours fast threads (the ones the instructions of which stay shorter in the pipeline), avoids starvation, and provides a good distribution of instructions from all threads.

Branch prediction mechanisms are not modified (branch target buffer (BTB) and pattern history table (Hwa93)), even if program counters and subroutine return stacks are replicated.

A larger register file is shared among all the hardware context. 32 registers per threads are available, plus 100 integer and 100 floating-point renaming registers. The renaming mechanism is able to map architectural registers of running threads to physical registers, avoiding conflicts. Once suitably *renamed*, instructions can be stored into the instruction queues, and then be reordered, scheduled and executed *without any further control*.

In order to have a high clock frequency, two more stages are added to the pipeline, as the larger register file needs more time

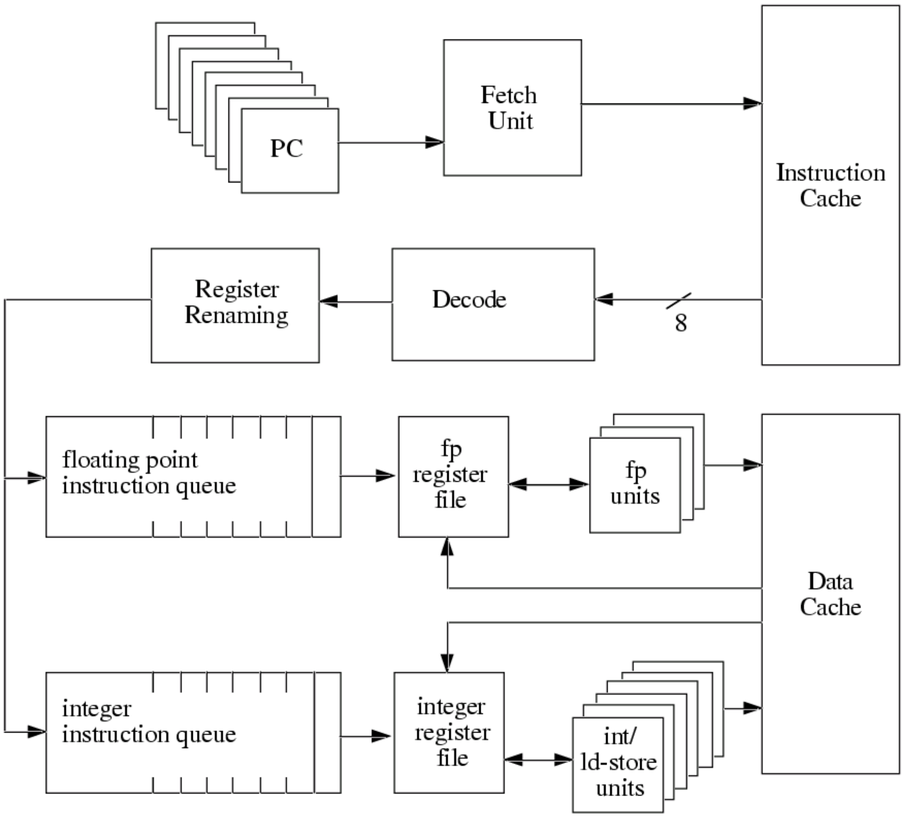
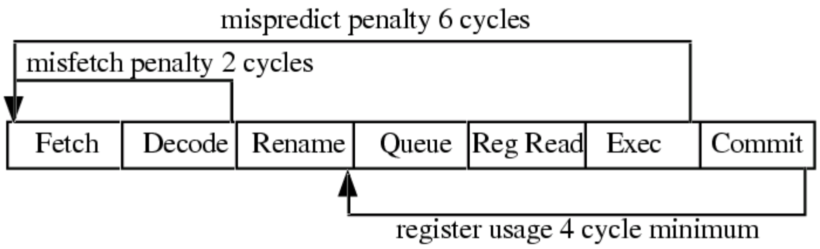
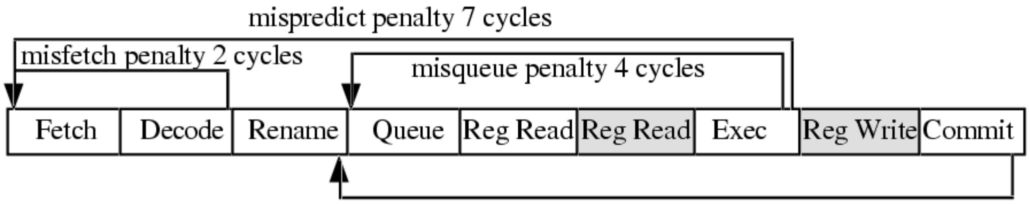


Figure 1.3: Basic SMT architecture (from MIPS R10000)



Original MIPS R10000 pipeline



Basic SMT architecture pipeline

Figure 1.4: Superscalar and SMT pipeline

to be read and written (see figure 1.4). This longer pipeline introduces, according to (LEE+97), a degradation of only 2% when a single thread is running.

Other changes are needed to implement SMT: per-thread instruction retirement and trap mechanisms, and an additional thread-identifier field in each BTB entry.

As explained, in this architecture only few structures are replicated, i.e. statically assigned to single threads, namely program counters, subroutine return stacks, retirement and trap mechanisms. All the other structures (functional units, caches, register file, translation look-aside buffer, branch prediction mechanisms) are dynamically shared among threads, and all available even when sequential code (only one thread) is being executed. The advantage of SMT can be so reached, keeping intact most of superscalar peak performance with sequential code. In figure 1.5 (Tul96a), performance comparison between a superscalar and the SMT processor under *Spec92* benchmarks work-load is shown. In the cited work, two version of the SMT processor are discussed: one basic, which features the most minimal structures to implement SMT (e.g. fetching from just one thread), and one advanced, which features all the solutions previously discussed.

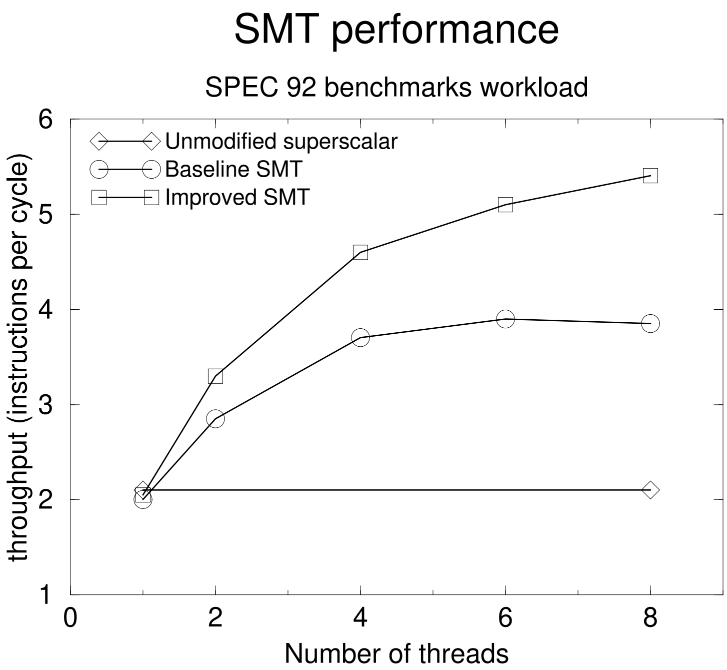


Figure 1.5: Performance of SMT (basic and advanced version)

1.2.2 Fast fine-grained synchronization

The unique availability of shared caches and register file offers the opportunity to implement very fast fine-grained synchronization among threads, without accessing the main memory, locally inside the chip. In (TLEL99), a hardware-based blocking-lock mechanism is described, that can be implemented into the SMT processor, offering some important features:

- high performance, because synchronization stays in the lowest levels of the memory hierarchy;
- resource-conservative, because passive waiting is implemented: waiting threads are waken up by the hardware;
- deadlock-free, because suitable ordering is introduced among threads,
- easy to build, as a very simple table can describe threads' waiting status.

The mechanism is implemented with a small structure associated with some (two in our simulation) functional units, called *lock box*. It stores one entry per hardware context describing the address of the lock (with a tag describing its validity), and a pointer to the `lock` instruction that blocked the thread.

Three new instructions (`acquire`, `release`, `try-acquire`) are needed to interface this mechanism. Their semantics is straightforward. More powerful functionalities, such as the *barriers*, are implemented in software. The interested reader is invited to look for more information (implementation, comparison with other mechanisms...) in the work cited above.

1.2.3 The simulator

In my work, extensive usage of the SMTSIM simulator (see (Tul96a)) has been done. This program is able to simulate execution of multithreaded programs and of multiple sequential programs, reporting detailed information as total completion time, average memory-access delay, number of register and functional-unit conflicts, and so on. It is also able to simulate the fast on-chip synchronization mechanism described above.

More considerations about SMT motivation, architecture, design-space and performance can be found in (TEL95, Tul96b, LEE$^+$97).

1.3 Overview of the thesis

Main goal of this thesis is a deeper understanding of what an advanced compiler can do to enhance performance on a SMT processor. In this direction, two main studies have been carried on.

The first one focused on the effectiveness of some advanced compiling techniques for SMT. To improve performance, the kernels from the Livermore loops benchmark have been rewritten with some standard techniques: the effect of this work and the generality of the proposed methodology are discussed with detailed experimental data.

The second one focused on modeling the performance of simple kernels running on SMT. The presented model is able to determine performance upper and lower bounds for a large class of short kernels. Experimental results collected in the first part are here compared against the expected figures. The model performs really well with small loops (less than 40 instructions): in this case the performance (useful processor utilization) is not farther than 5% from the maximum expected value.

The thesis is organized as follows. Chapter 2 discusses shortly some previous results about SMT, such as performance results and threaded multiple-path execution. It also presents shortly the engineering problems offered by SMT, and describes the *Cray MTA* architecture, the compiler of which is used as a term of comparison throughout the thesis.

Chapter 3 offers some considerations about some common parallel techniques, that are studied in the SMT context. Experimental results collected by MT versions of Livermore loops running on the SMT, and the techniques used to improve their performance are here discussed in their global aspects. Detailed results are instead shown in chapter 4. Chapter 5 introduces and describes an original performance model for the SMT processor. Chapter 6 concludes and presents some open questions and future works.

An appendix reporting the most used acronyms closes this work.

Chapter 2

Related work

In this chapter, a rapid overview of some interesting results about SMT is given. Last section gives also a quick description of *Cray MTA* architecture, that is used as a term of comparison throughout the whole thesis.

2.1 Threaded Multiple Path Execution

In (WCT98) and (WTC99) an interesting usage of SMT capabilities is discussed, called *threaded multiple path execution* (TME). This technique uses idle contexts on SMT to execute speculatively multiple paths of execution.

This technique tries to face the problem posed by branch misprediction, that is growing worse with the increasing execution bandwidth and the lengthening of functional pipelines. TME consists in using idle contexts on a SMT to aid branch prediction. This is a perfect match with the architecture, because unused resources are offered to improve the overall performance: when TLP is low, the idle contexts are used to try to increase ILP. When facing a branch, execution continues on both paths, on different contexts. When branch is resolved, the thread executing the wrong path is simply discarded, and becomes free for a new computation.

Some peculiar features of SMT make this solution particularly efficient. As register file is shared, there is no need to copy all the registers to start a new execution path: copying register map, much smaller, is enough.

The TME implementation discussed in the cited work is able to identify good candidates for spawning, to start alternate-path

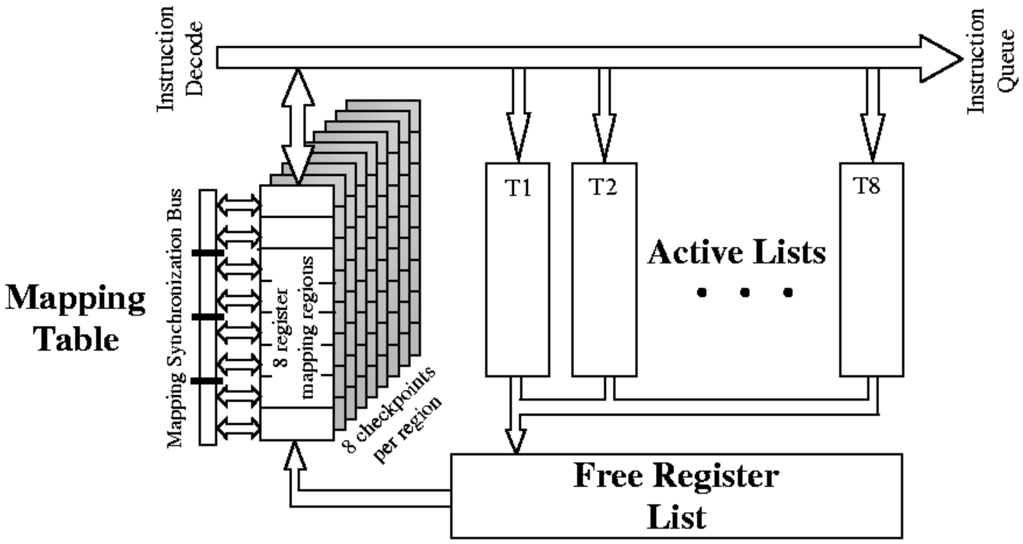


Figure 2.1: SMT architecture improved with TME capabilities

thread in a separate hardware context, and to provide efficient instruction fetching for both paths.

A new structure, called *mapping synchronization bus* (MSB), is introduced into the architecture (see figure 2.1), with the role of copying register mapping among different threads, and keeping their maps updated when alternate-path threads are spawned onto the idle harware-context.

The authors show that TME achieves an average 14% single-program performance improvement, considering a branch misprediction penalty of 7 cycles. TME seems so to be a good approach to increase ILP for programs with high-misprediction rate on SMT.

2.2 ILP vs. TLP

Compiling issues are a very hot topic. Previous work on this includes (CT99, TLEL99, RCT$^+$99, LEE$^+$97). In the direction of my work, very interesting is especially (MCFT99). In this work, two main points are discussed: differences in performance between ILP and TLP on a SMT, and a way to model performance for MT code.

As said, SMT makes TLP operationally equivalent to ILP. The main

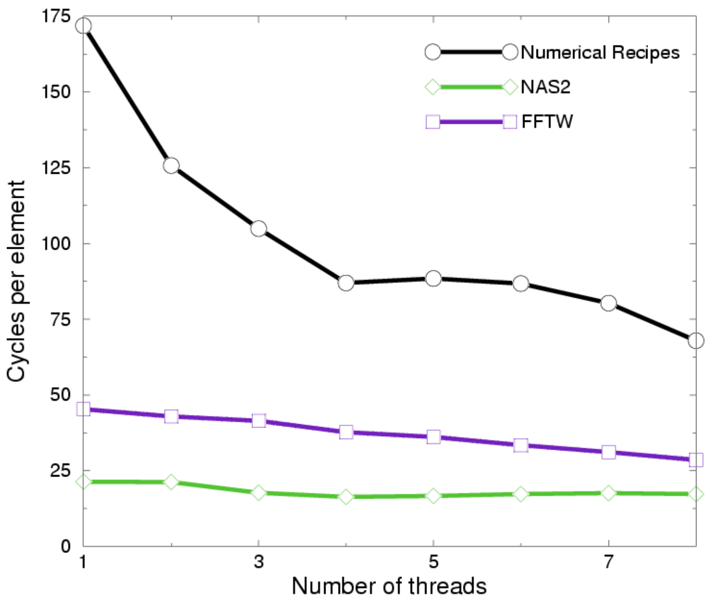


Figure 2.2: Comparison of FFT algorithms on SMT in (MCFT99)

question of the paper is if SMT performs equally well, when the independence comes from ILP or TLP. In the paper, three case studies are analyzed: matrix multiply, fast Fourier transform (FFT) and integer sort.

As an example, results on fast Fourier transform are here reported. Some different algorithm, that compute eight one-dimensional Fast Fourier Transform, with 2^{20} points, were compared: a naïve (Numerical Recipes in C) implementation, parallelized[1] distributing one-dimensional (1D) FFTs cyclically; FFTW, parallelized distributing 1D FFTs cyclically; NAS FT, parallelized interleaving outer loop.

Very good (2.53x) speedup has been observed for the naïve algorithm, a little smaller speedup for the other two (1.3x for FFTW, 1.59x for NAS FT). Nonetheless, the best performance is reached by NAS FFT. It can be said, as a general principle, that *a multithreaded poor algorithm is not a substitute for a good algorithm, single- or multithreaded* (see figure 2.2).

[1]The awful term *to parallelize* appears another 37 times throughout this work: I apologize since now for that. Suggestion for a nicer periphrasis are more than welcome.

	nt	f1	f2	f3
MM	-0.16	0.60	-0.06	0.43
FFT	-0.11	-0.08	0.48	0.89
IS	-0.02	0.06	0.65	0.66

Table 2.1: Correlation of factors with performance

Probably the most interesting result in the paper is anyway the performance model presented in the last part. The authors try to predict performance of different implementation from few parameters that are recorded from the SMT simulator.

These parameters are:

- nt: number of threads,
- $f1$: register locality (measured as communication to computation ratio),
- $f2$: cache and TLB locality (measured as average access time),
- $f3$: demand of resources (measured as conflict rate),
- $f5$: number of instruction of the implementation.

Roughly, it can be said that ILP increases the figures for $f3$ and $f5$, and it is good if $f1$ and $f2$ are high. On the other hand, TLP affects $f2$, and it is good when $f1$ and $f2$ are low.

In the paper, Brewer's technique (Bre95) is used to determine automatically a model for performance, as a weighted sum of the factors, for every problem. In table 2.1, coefficient for every problem, as reported by the authors, can be found.

The very good quality of modeling (see figure 2.3), and the fact that the role of coefficient is radically different for every problem, should be observed. Even if the result is very good and encouraging, in my opinion a more general model, able to describe performance of different problems, and not depending from profiling information, is still needed. This is what my thesis tries to do.

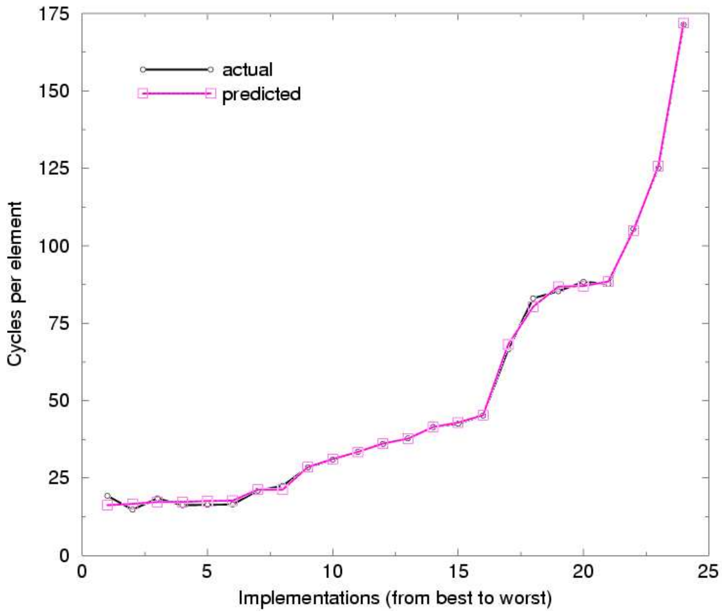


Figure 2.3: Accuracy of modeling FFT in (MCFT99)

2.3 Explorations in symbiosis

In (SMC+99), Snavely *et al.* compare *Cray MTA* and SMT in their *symbiosis* opportunities. With this term, the authors refer to the increase in throughput that can occurr when two or more applications are executed concurrently on a MT computer.

SMT and MTA share functional units among different threads: if some resources are left idle by one thread, they can be used by another, possibly without affecting the first one. Positive symbiosis is actually observed under many conditions by the authors, that suggest that a good co-scheduling (an accurate choice of which programs are to be executed together) could greatly improve the global throughput.

2.4 Does SMT pull its weight?

SMT is a smart technique able to exploit TLP in a modified superscalar processor. Even if this new feature can be added with a little overhead, this additional chip area could be used to add different structures, as additional functional units, larger caches, or to implement a multiprocessor on a single chip.

Many researchers, even if trusting the opportunities offered by SMT, are still evaluating if this is actually the best architecture for next-generation chips (DeH00a, Kub00, Ama00). Nonetheless, a strong commercial interest to SMT can be seen (Die99).

An interesting work by Burns and Gaudiot (BG00) analyzes the area usage that SMT would introduce in a regular superscalar processor. Their work studies how to transform a *MIPS R10000* into a *R10000-2x* (with twice as much functional units), and then into a SMT processor.

In the paper, the original *MIPS R10000*, built with a 1996 0.35 µmtechnology, is extrapolated to a 2000 0.18 µmtechnology, that allows to have a dispatch width (in the paper with the meaning of maximum sustained processor throughput) twice as large. The entire pipeline is consequently increased.

A redesigned memory hierarchy is used in the simulated *R10000-2x*: a 256KB level-2 cache is added, the original 16MB level-2 cache becomes a level-3 cache, a larger (4x) TLB is featured, an improved branch prediction mechanism is added (Gshare unit).

SMT features are then added: replicated instruction counters, status registers and return address stack, and a larger register file are needed. A slightly modified *icount* policy is implemented (TEE$^+$96).

To keep the clock rate fast enough, processor's pipeline needs to be modified with respect to the original architecture, as explained in section 1.2.1.

Detailed simulation of different architectures was done by the authors. The design space included different fetch capabilities (fetching one or more cache lines, with merging capability), dispatch width and number of threads. In the following paragraphs, this notation will be used: fX is the number of different threads fetched per cycle, tY is the maximum number of threads, dZ is the dispatch width.

Their results (based on *Spec95* benchmarks) shows that the increased fetch capability affects the single thread performance (longer pipeline), but is convenient if more than one thread is running, as it reduces the fetch bottleneck. In table 2.2, results for two architecture with dispatch width equal to 8, and fetching from 1 and 2 different threads each cycle respectively, are shown.

In the second part of their paper, very detailed results about the layout overhead of SMT are reported. Burns and Gaudiot give an

chip. config	number of threads 1	2	3	4	5	6	7	8
fetch 1 th.	2.19	3.20	3.83	4.18	4.43	4.59	4.69	4.75
fetch 2 th.	2.11	3.22	3.96	4.35	4.70	4.91	5.12	5.19

Table 2.2: Simulation results in (BG00)

functional block	R10000-2x	SMT
remapping tables (register renaming)	$O(d^2)$	$O(t * \log_2(t))$
enlarged INT and FP register files	$O(d^2)$	$O(t)$
fetch block, multiple PC and *icount* fetch policy	$O(d)$	$O(t)$
branch predict block, return stack	$O(d)$	$O(t)$
instruction squash (O^3)	N/A	$O(t)$
instruction commit (O^3)	$O(d^2)$	$O(t)$
free list with wider tag storage (O^3)	$O(d^2)$	$O(\log_2(t))$
instruction queue (O^3)	$O(d^2)$	$O(\log_2(t))$
recover *icount* (O^3)	N/A	$O(\log_2(t))$
dtag, itag	$O(\log_2(d))$	$O(\log_2(t))$
routing	$O(d)$	$O(\log_2(t))$

Table 2.3: Layout blocks with large SMT overhead

estimation of the overhead caused by every functional block, measuring commercial transistors/interconnect level layouts and extrapolating results to larger structures. Their results are reported in table 2.4: the cost of building *R10000-2x* starting from *R10000* is measured in the second column as a function of the dispatch width d; in the third column, the overhead introduced by SMT features, as a function of the number of threads t, can be found (O^3 standing for out-of-order execution). In most cases, the former grows with the square of the dispatch width ($O(d^2)$), the latter instead linearly with the number of threads ($O(t)$).

Other blocks are affected indirectly by SMT: the increased throughput creates new bottlenecks, that require new original solutions. Even if these added structures are not strictly required by SMT, SMT takes great advantage by their introductions. An example is given by the fetch unit, that becomes a critical point in SMT. Fetching multiple cache lines solves this problem, and improves per-

functional block	R10000-2x	SMT
INT and FP arithmetic units	$O(d^2)$	$O(1)$
Dcache (additional ports and 2x cache)	$O(d)$	$O(1)$
Miscellaneous	$O(1)$	$O(1)$

Table 2.4: Layout blocks unaffected by SMT

formance (TEE$^+$96). This solution requires improved Icache, TLB, and branch prediction mechanisms, that need multiple read ports. New structures are also required to merge the fetched lines into one.

Other structures instead are unaffected by SMT features, but grows when moving to *R10000-2x*, as shown in table 2.4.

The authors consider these parameters to give an estimation of the per cent area increase for the two new configurations discussed, with respect to three metrics: processor core (that includes pipeline stages and logic, and is critical to determine clock rate and overall design cost), entire chip minus L2 cache (an important metric, as caches can quickly be changed with improved technology), and entire chip (that is a good metric of packaging cost and power dissipation).

As the reader can see in table 2.5, fetching unit is considerably larger in SMT with respect to R10000-2x, but the larger core-area increases are given by O^3 execution logic and register files. Nonetheless routing and L2 cache, that occupy alone 46.5% of chip area do not grow with SMT. Icache is another source of area overhead, due to multiple read ports. In figure 2.4, total chip area required by different implementations can be found.

The two authors also perform a comparison with other opportunities, that have similar chip area (table 2.6), trading high number of hardware contexts with other functional blocks: a four times larger branch unit (SMT.f2.t6.d8_BU4x), larger cache (SMT.f1.t4.d8_cache), larger dispatch width with two threads (SMT.f2.t4.d10), larger superscalar (SMT.f1.t1.d11). Figure 2.5 suggest that the best throughput is reached by the eight-thread versions.

The authors conclude confirmings their trust in SMT:

- SMT overhead is equal to 5.8% of the core and 3.7% of the entire processor with 2 threads, 46.7% and 28.3% respectively with 8 threads;

Function	Chip block	MIPS R10000-2x in 0.18μm		Relative (%) area increase of adding SMT			
		area (mm^2)	area (mm^2) w/ SMT	to R10000-2x block	vs. core area	vs. chip area w/o L2 cache	vs. chip area w/ L2 cache
Dcache	Dcache	11.4	11.4	0		0	0
	Dtag	1.6	1.6				
Icache	Icache	9.1	13.7	**50**		2.9	2.2
	Itag	1.3	1.9				
TLB	TLB	4.4	5.7	30		0.7	0.6
Fetch	Fetch	1.0	4.6	**157**	5.6	4.1	3.1
	Bpred	3.6	7.2				
Decode	Decode	2.3	4.5	96	1.7	1.2	0.9
O^3 execution	Remap-logic	2.5	3.3	68	**19.4**	13.9	10.6
	Remap-tables	2.4	16.2				
	FreeList	2.3	16.2				
	IQ	7.0	8.8				
	LSQ	9.4	11.8				
	FPQ	6.3	8.0				
	Reorder	6.1	7.8				
	RAS	0.3	2.1				
Register Files	Int.RF	5.7	18.8	**231**	**20**	14.3	10.9
	FP.RF	5.3	17.6				
Arith. units	Int FU	7.6	7.6	0	0	0	0
	RPMUL	4	4				
	FPALU	8.3	8.3				
Misc.	ExtInt	5.2	5.2	0		0	0
	JTAG	0.9	0.9				
	Misc.	2.4	2.4				
	I/O	13.7	13.7				
Routing	Routing	**52.7**	**52.7**	**0**	0	0	0
256K L2 cache		**55**	**55**	**0**			0
Total	core	126.7	188.8		46.7		
	w/o L2 cache	176.7	242.7			37.1	
	w/ L2 cache	231.7	297.7				28.3

Table 2.5: Increased area of R10000-2x and SMT

Configuration	Chip area (mm^2)
SMT.f1.t8.d8	284.8
SMT.f2.t8.d8	297.7
SMT.f2.t6.d8_BU4x	304.4
SMT.f1.t4.d8_cache	338.5
SMT.f1.t2.d10	319.1
SMT.f1.t1.d11	316.1

Table 2.6: Alternative micro-architecture with similar area

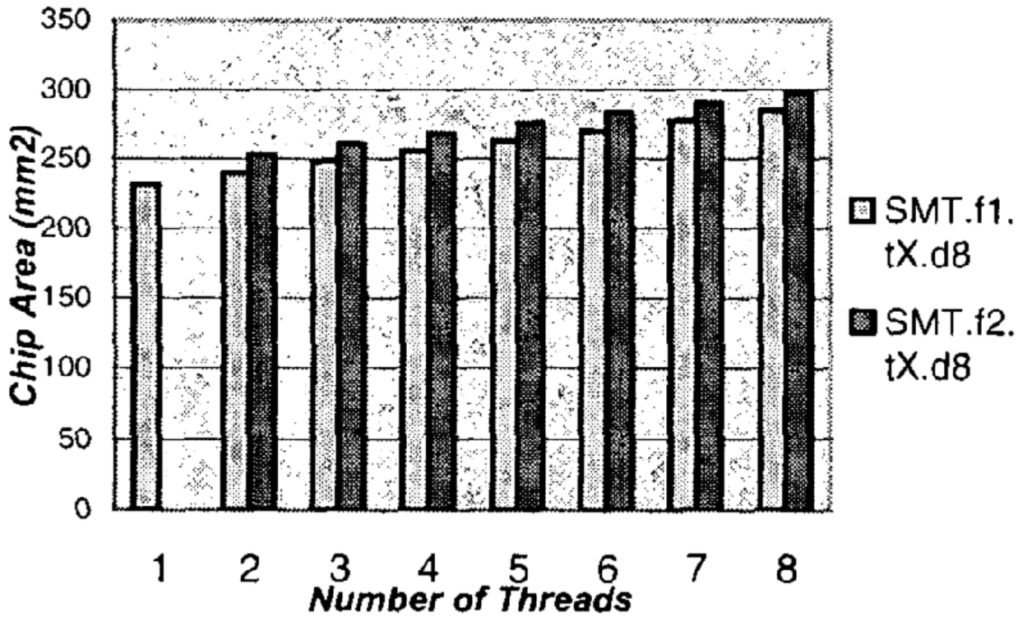


Figure 2.4: Area overhead of different SMT implementations

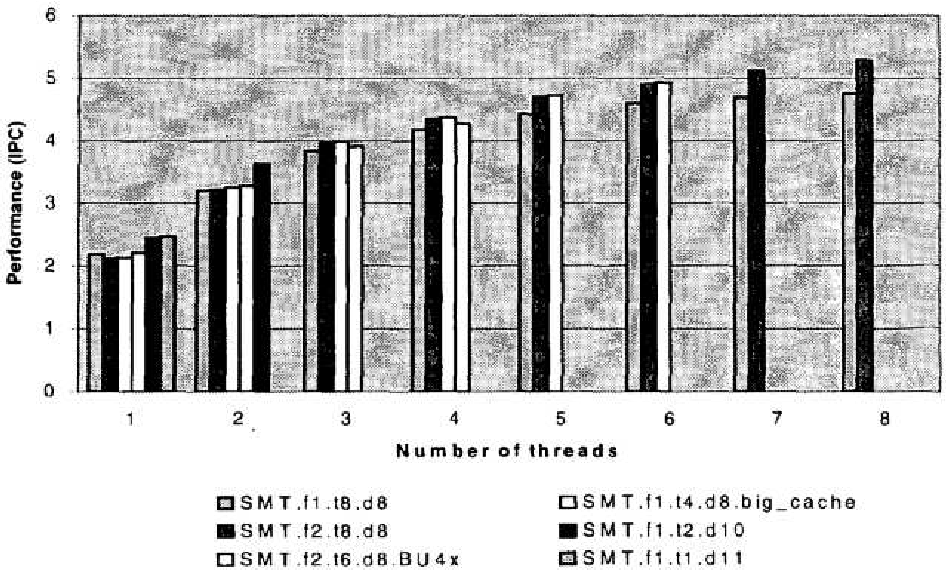


Figure 2.5: Performance of different architectures

- SMT throughput increase is about 142%, really substantial when compared with area requirements;
- SMT throughput increase is about 114% with respect to a superscalar processor, with larger execution bandwith, requiring the same or larger area.

2.5 Other works about SMT

I would like also to cite some other interesting recent works on SMT.

Collins *et al.*, in (CT99), describe a hardware support that classify different types of cache misses, simulated on top of SMT. This knowledge is, in the paper, applied successfully to many problems, among which cache prefetching.

Tullsen *et al.*, in (TS99), present a low-overhead technique for register value prediction, based on register-value reuse locality. Their results show a 12% increase in performance.

Lo *et al.*, in (LPE$^+$), discusses a software-directed approach to register deallocation. Their technique allows a more efficient register file utilization in SMT: their results shows that a 264 register files, with

Figure 2.6: Cray MTA machine at the San Diego Supercomputing Center

software deallocation, can perform as well as a 352 register files, managed in the traditional way.

2.6 Fine-grained multithreading: the example of MTA

As described in the introduction, SMT overcomes the concept of fine-grained multithreading (FGMT), moving to the new described solution. FGMT is nonetheless a still vital approach to advanced microprocessor architecture, that is featured by the *Cray MTA* (formerly *Tera*), that represents the state of the art for von Neumann FGMT architectures.

MTA is the result of a long-planned project, begun in 1990, in the reach for performance, with three main goals (ACCK90):

- very high speed implementation (fast processors (HK97) and high scalability);
- general-purpose applicability;
- easy compiler implementation.

In order to reach them, many innovative solutions are proposed by *MTA*:

- a custom GaAs integrated circuit (that will be soon replaced by a more standard CMOS technology, that has reached the needed performance);
- uniformly accessible memory (UMA);
- absence of caches;

and, as said, a highly efficient support to FGMT. Every processor is able to execute up to 128 threads, interleaving them with FGMT: it is able to fetch one VLIW instruction from a different thread, every cycle, with a context-change cost equal to zero. This allows to hide very long memory latencies (up to 1024 cycles) and any pipeline delay (branch prediction, data dependences...): to programmer's point of view, results of any instruction are always available in the next cycle.

Result of a careful hardware-software cooperation design (ABC^{+}97), *MTA* is a highly programmable machine: multithreading is very easy to achieve, and parallel programming comes out to be quite easy. Full support to thread creation and removal, and to lightweight synchronization, makes its programming straightforward: programmer do not have to think about data allocation in memory, cache management, structural hazards.

Present in just one copy in the world, at the San Diego Supercomputing Center, its performance is still being evaluated. Results are quite encouraging, even if the actual configuration suffers often from hardware problems. I was offered the opportunity to play a bit with its compiler, that can extract automatically parallelism from most programs. Implemented techniques include: loop unrolling, loop interchanging, cyclic reduction, loop fusion. An advanced tool, called *canal*, offers some feedback about the solutions utilized in the parallelizing process, so that programmers can further help the compiler in its task.

Chapter 3

Tuning TLP on Livermore Loops

3.1 Introduction

The work here presented has the main goal of developing a better and deeper understanding of the complex iterations that can be found among threads running on the SMT processor. It also wants to determine what an advanced compiler can do to improve MT performance. In this direction, a thorough analysis of the behavior of some standard kernels has been done. In this research, the Livermore loops have been chosen as a representative case study.

The Livermore loops are a set of 24 heavy-computing short kernels, that can be found as main core of many important scientific programs. Main features are a very high demand for processor resources and very different behaviors: actually, they span from indepent-iterations loops (highly parallel) to other ones with tight dependences, and offer cache and functional units utilization patterns broadly different (McM86).

They were developed to *measure numerical computation rates for a spectrum of cpu-limited computational structures or benchmarks* (F. H. McMahon, 1972). They are so particularly interesting to evaluate different architectures' performance with a typical scientific work-load. Furthermore, peculiar characteristics of every single kernel make them a real challenge to compilers, that implement more and more complex techniques to improve compiled code's efficiency. Actually, in this research, I played the role of a compiler, trying to understand what the limits to advanced compiling for SMT can be.

Originally written in *Fortran*, a *C* version is also available. I used

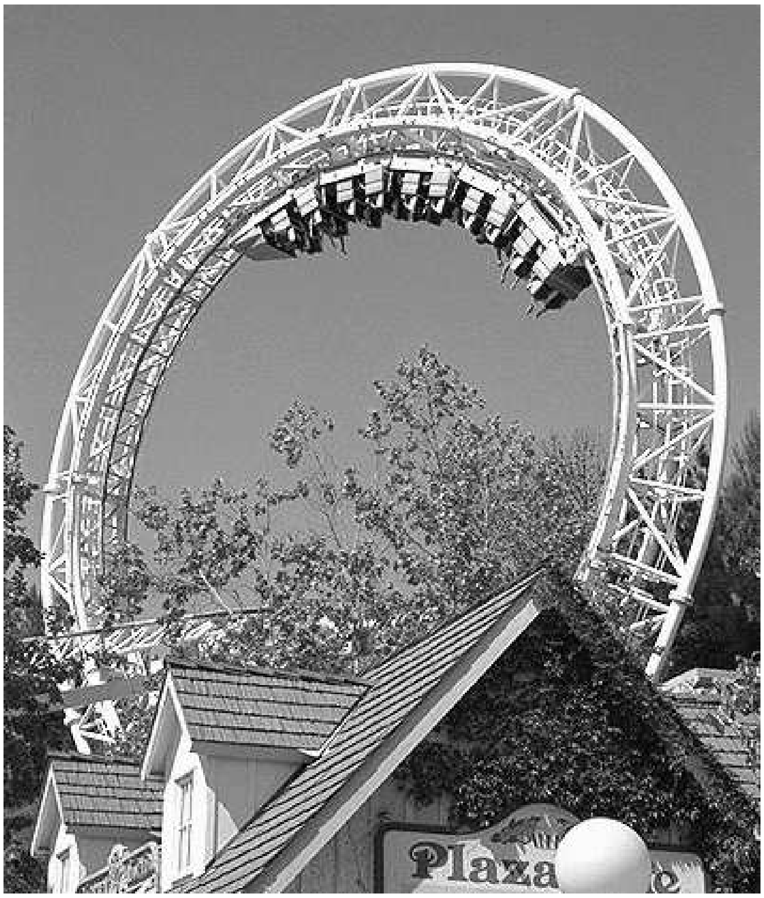


Figure 3.1: An interesting example of loop (the first ever looping roller-coaster, the Revolution at Six Flags, Los Angeles, CA)

the original *C* versions as closely as possible. In a rather unexpected way, after some programs were slightly restructured to make them easier to parallelize (some of them still featured explicit jumps with `goto`!), they performed more efficiently. In these cases, I preferred to use my new versions, as more respectful of what present program should be.

I wrote and run many different multithreaded versions of every loop. Main goal of this part of my study was to understand which techniques can be used to improve performance of these kernels on SMT, and to develop some SMT-specific techniques, that could be used by an advanced compiler. More than smart tricks for every single kernel, I tried to use what is considered to be standard

	ICache	DCache	L2 Cache	L3 Cache
Size	64KB	64KB	256KB	2MB
Associativity	DM[a]	DM	4-way	DM
Line size	32	32	32	32
Banks	8	8	4	1
Transfer time/bank (cycles)	1	1	2	2

Table 3.1: Details of memory hierarchy

[a]Direct mapping.

in modern compilers: common techniques as cyclic reduction and loop fusion were exploited, while kernel-specific solutions, based on a higher level of semantics, were carefully avoided. Again, *the goal was to test some general techniques that could be useful with SMT, not to exploit the best performance for each kernel*.

As a term of comparison, particularly interesting was the usage of the *Cray MTA compiler*, kindly allowed by the San Diego Supercomputing Center, where the first running *MTA* machine can be found. It is my opinion that an advanced compiler for the SMT processor will be probably similar to it in many aspects, even if the presence of advanced superscalar features introduces lots of problems in the compiling process (as discussed in section 3.7.2).

3.2 Methodology

Main features of the simulated SMT processor are reported in tables 3.1, 3.2 and 3.3. The *Atom* column describes the parameters used by *Atom* to compute the critical path scheduling (see section 5.1). As explained in the founding work by Tullsen *et al.* (TEL95), these parameters describe what a SMT built in the next future will probably be.

All the kernels were rewritten manually. I tried not to introduce higher-level knowledge of program semantics, with the goal to behave as close as possible like a compiler. Actually, I try to confine myself to observe dependences among iterations, using general methods (GCD, diophantine equations...), and to introduce the smallest number of synchronizations and barriers, needed to guarantee semantics: my rewriting work was led every time by general

Instruction class	Latency SMTSIM	Latency Atom
integer multiply	8/16[a]	8/16
conditional move	2	2
compare	0	0
other integer	1	1
FP divide	17/30	17/30
other FP	4	4
load(L1 cache hit, no bank conflicts)	2	2[b]
load (L2 cache hit)	8	-
load (L3 cache hit)	14	-
load (memory)	50	2
store	1	1
control hazard (predicted)	1	1
control hazard (mispredicted)	6	1

Table 3.2: Latency of instructions in the SMTSIM simulator and in the Atom tools

[a]Latencies depend from instruction's precision.

[b]This value is usually set to the average memory-access delay, as measured by the simulator.

L1-L2 bus latency	2 cycles
Instruction TLB	48 entries
Data TLB	128 entries
Integer renaming registers	100
FP renaming registers	100
Fetch bandwidth	8 inst. per cycle
Functional units	6 integer, 3 FP
Integer units	4 load/store, 2 synch. out of 6
Maximum number of threads	8
Fetch policy	Icount, 1:8 [a]
Integer instruction queue	32 entries
FP instruction queue	32 entries

Table 3.3: Parameters of the simulation

[a]For more details on fetch policy see (Tul96b).

principles, though, and sometimes I had to be very *conservative*.

Parallelization techniques used includes: interleaving, loop fusion, cyclic reduction, loop peeling, invariant motion, local accumulation (described in section 3.5). All of these seem to be very general and useful in most cases. This work offered a good insight of which information can be collected by the compiler in order to perform good parallelization.

In order to simplify the discussion, the kernels will be classified into four groups:

- independent-iteration loops,
- loop-carried-dependence loops,
- accumulation loops,
- larger loops.

Some common features can be found among the loops belonging to the same group, even if unexpected behaviors have been observed. In the following sections, two main figures will be discussed: processor utilization and completion time. Even if completion time is the most important value when discussing the effect of multithreading, processor utilization is a useful description of how well processor resources are exploited: we can not expect very high improving in terms of completion time, if processor utilization is high for the sequential version.

In this chapter, only general considerations for every group will be made. The interested reader can find more details in the next chapter, such as all the experimental data, and a short discussion describing every kernel and its parallelization.

In the following plots, these two figures — processor utilization and completion time — are described by two curves: the former is drawn with circles, the second with squares. Both of them are shown as per cent values: completion time is normalized with respect to the sequential completion time (let equal to 100), and processor utilization as average percentage of used *slots*. X axis represents the number of threads (with 0 standing for the sequential version), Y axis represents the values per cent of utilization and completion time. Differences between the sequential and the one-thread versions describe the cost of restructuring the loops.

Compilation flags and completion time (cycles) of the sequential code, are reported for every loop. All the kernels were compiled with the *GNU C compiler* (*gcc*).

3.3 Independent iterations

This group collects all the loops that feature independent iterations (kernels no. 1, 7, 8, 9, 10, 12, 15, 17, 21, 22). They express vector computations, that can be carried on independently for every element. The only actual dependence is given by induction variables, which are local to every thread in the MT versions. As shown in (LEL+97), in these cases *iterations interleaving* among threads (see program fragment 3.1) is the most efficient way to express parallelism: it has been shown that a better cache utilization is reached with this solution. This kind of loops is easily run on the SMT processor, with good speedup: these loops generally present an asymptotic increasing of performance (both shown metrics), and reach a good processor utilization (i.e. further improving is limited), about 65% on average. See for instance figure 4.2.

Some general principles can be learnt from the collected data: when iterations are independent, they easily become TLP on SMT; if sequential code has low processor utilization, this TLP can contribute to a major improving, while kernels with high initial processor utilization show small or no improvement.

3.4 Loop-carried-dependence loops

In this group, loops featuring loop-carried data dependences are collected (kernels no. 5, 11, 19, 20, 23). Interleaving is not useful here, as iterations need to be executed strictly in order: limited opportunity for increasing ILP is offered.

Nonetheless, one of them (kernel no. 11) was successfully parallelized, introducing the so-called *cyclic reduction* (see section 4.2 on page 52), a powerful algorithm for the running sum problem. This technique can be applied even to kernel no. 5, that present some common points, but this was not tried due to time limits.

A very important fact is that this type of reductions can be recognized automatically by the compiler, that can apply techniques

(a)

```
for ( k=0 ; k<n ; k++)
{
    a[k] = ....
}
```

(b)

```
for ( k=threadid ; k<n ; k += NUMTHREADS )
{
    a[k] = ...
}
```

Program fragment 3.1: Original code (a) and MT version with interleaved iterations (b)

as the one just cited. Actually, the *Cray MTA compiler* already does this kind of optimization, and it is my opinion that a suitable set of implementation templates needs to be developed and introduced into a smart compiler for the SMT.

Another aggressive technique (loop skewing) has been used for kernel no. 23, that updates a matrix in a simple row-column order, but the overhead of this solution prevented any performance improvement. I do not exclude that a better tuning could improve performance, as processor utilization is really low in the sequential version.

Kernel no. 5 and 19 have been parallelized introducing some lock primitives around the critical variable updates, hoping that the loop overhead (test, increment...) could be partially hidden. This is not the case, as synchronizations introduce a very high overhead, that is not followed by any increasing in TLP.

3.5 Accumulation loops

The following loops (kernels no. 3, 4, 6, 13) are particularly interesting, as they feature some independent computation, followed by accumulation of all the values. So, one part of the body could be easily parallelized among threads (interleaving), while accumulation was protected by suitable (ordering) locking to prevent critical races.

In many cases (kernel no. 3, 4 and 6), a general technique that I will call *local accumulation* was useful. It implements a simple consideration: if the accumulation is carried on by an associative and commutative function, the order is not important. In this case, every thread can compute a local summation, that at the end takes part in the global sum, performed by one specific thread[1].

This kind of loops generally has good improvement with few threads. After this, the sequential ordering introduced by locking makes added threads useless (see for instance figure 4.16).

3.6 Larger loops

Under this denomination, all the other loops, featuring a larger loop-body and a more complex behavior, are collected (kernels no. 2, 14, 16, 18, 24). They represented a great challenge for the optimizing compiler (myself!) as their semantics was difficult to determine, due to complex branching and data-dependent memory-accesses. The common technique was to interleave iterations, introducing ordering locking (one thread wakes up the following one in the correct order) to protect possible dependences. In some cases, this is actually all that could be done for these very complex loops, where all the common techniques failed. In this case, it must be said that the compiler can not improve performance, due to limited knowledge of the code: the programmer should introduce e.g. some directives to help compiler in its task.

3.7 Further analyses

3.7.1 Running with 16 threads

Some tests have been run, with some representative loops, to see if much more (twice as much) hardware threads could be useful to improve performance. The simulator was modified by increasing the number of context, but not enlarging the caches, the number of renaming registers, the execution bandwidth and so on.

[1] I observed that a tree reduction for the second part of the algorithm had a too high overhead to be useful with SMT.

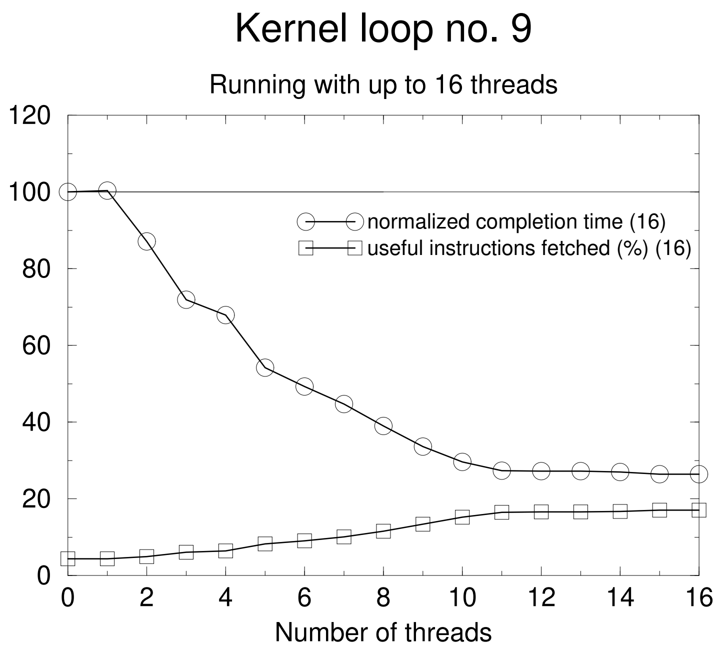


Figure 3.2: Execution statistics for kernel 9 (16 threads)

As the reader can easily see (figures 3.2, 3.3, 3.4, 3.5 and 3.6), the trends highlighted with up to 8 threads continue smoothly up to 16 threads (the first part of each curve is identical to the one shown in figures 4.4, 4.5, 4.7, 4.8 and 4.10). One strong point is that SMT shows no effects of thrashing or conflicts for resources: the increased memory latencies, due to a higher number of memory accesses, are perfectly hidden by the increased opportunities of TLP.

Another important point is that, even if some kernels (9 and 22) improve with a high number of threads, it seems that generally few (about 4) threads are needed to gain a good speedup. This kind of consideration is really important to trade the possible advantage with the cost of adding more hardware contexts.

3.7.2 Rescheduling basic blocks

In order to collect some information about SMT ISA, to be used in the modeling work, I spent some time hacking the assembler code produced by *gcc*, trying to determine bottlenecks and inefficiencies. My goal was to reschedule the most stressed basic blocks, in order

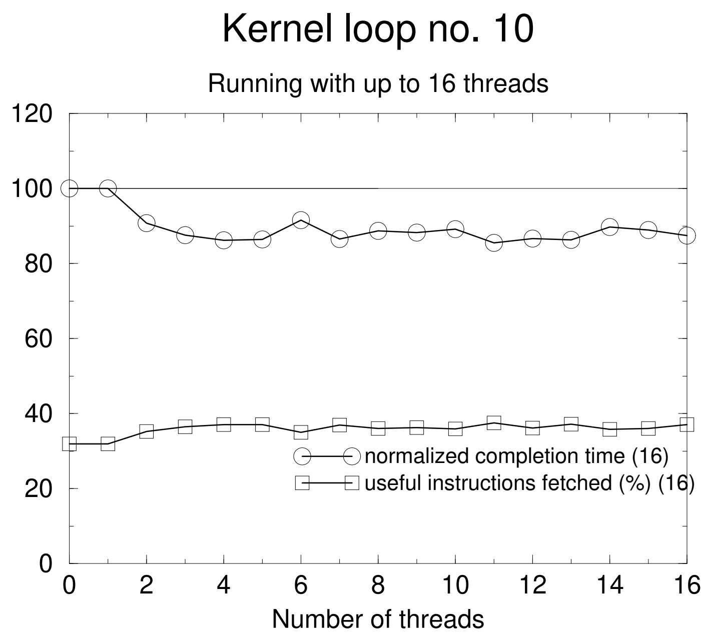


Figure 3.3: Execution statistics for kernel 10 (16 threads)

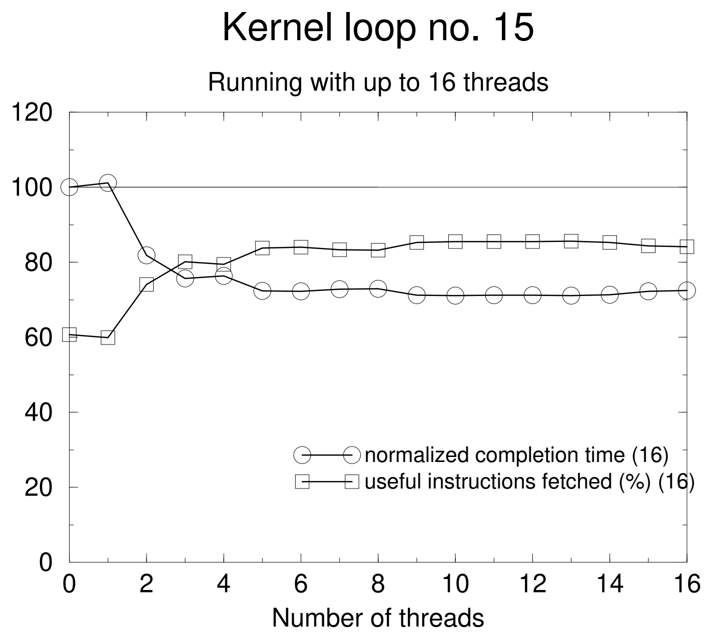


Figure 3.4: Execution statistics for kernel 15 (16 threads)

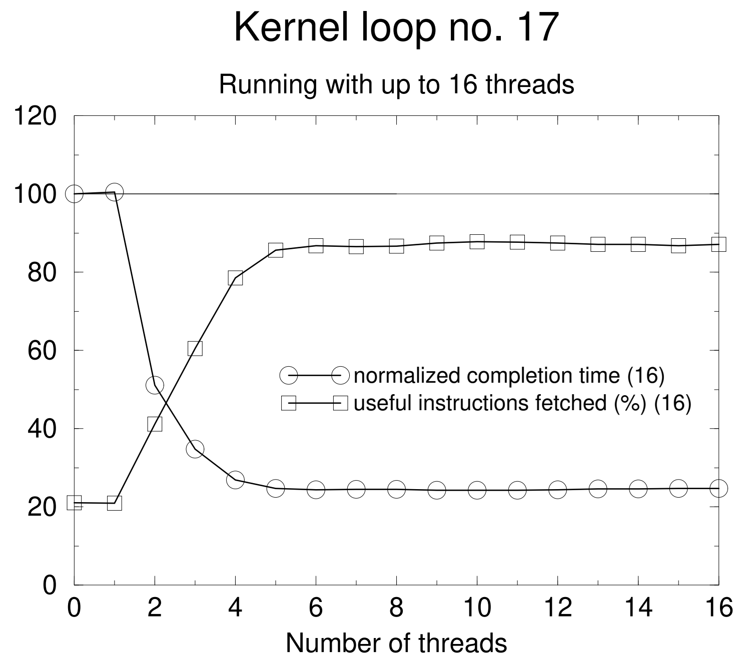


Figure 3.5: Execution statistics for kernel 17 (16 threads)

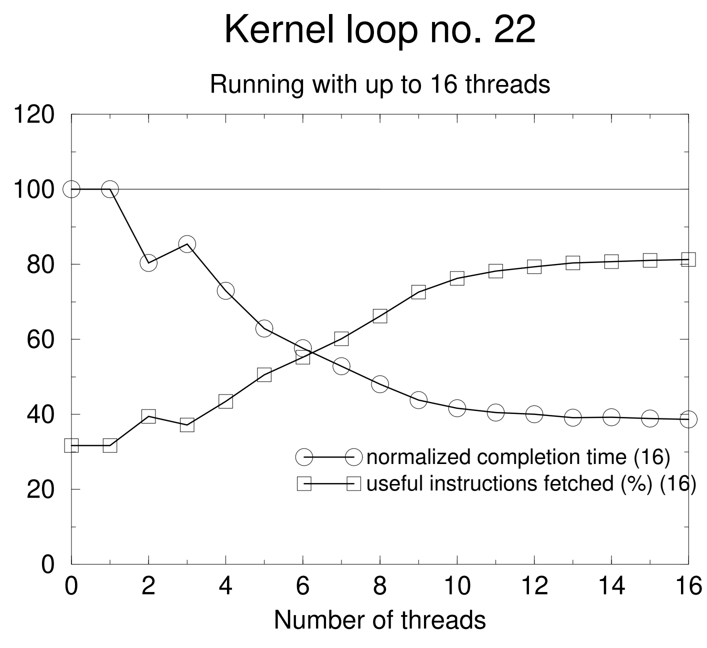


Figure 3.6: Execution statistics for kernel 22 (16 threads)

Kernel no. 9			
	gcc	opt	opt 2
1 thread	4584619	4574635	4574635
8 threads	1715602	1825738	1684350

Kernel no. 10			
	gcc	opt	opt 2
1 thread	177607	177255	178166
8 threads	155724	153422	N/A

Table 3.4: Experimental results for reschesuling

to improve performance figures. Particular attention to the most important architectural characteristics of the SMT processor has been paid:

- number and types of functional units,
- latency of instructions.

Some simple heuristics have been used: I tried to keep dependent instructions as far as needed (at least, as far as their latency), and to avoid code subsequences that would not fit the number and type of functional units. To do that, I reallocated data in registers (*gcc* is very *cheap* in using them), using more of them in order to be able to move instructions more freely.

Results are not encouraging: after a thorough scheduling, differences in performance are really small, and not always for the best. As a case study, the results of rescheduling for kernel no. 9 and 10 are described in table 3.4. They were chosen as they feature one single basic block that is executed more than 99.5% of the time (so to have larger changes with small code changes), and they have a radically different behavior when parallelized. *opt1* and *opt2* label two different versions, in which the number of used register was increased, and execution order was carefully considered.

The scheduling task is particularly difficult and can be useless or even pernicious. As a matter of fact, some advanced features of the simulated processor affects the actual behavior of the executed source code. In particular, some considerations should be done about register renaming and instruction reordering. Increasing the number of used architectural registers, trying to remove false (storage-related) dependences, is not very important, as this task is well performed by hardware: it can be useful to have more room for code-motion during the compiling process, but it's not so important for performance. Furthermore, far more registers (renaming registers) are available during execution, that can be used to hide this kind of dependences, better than every possible static data allocation.

Similarly, instruction scheduling in the source code is not important with a processor featuring advanced out-of-order execution: differences between scheduled and non-scheduled versions are smoothed by SMT run-time reordering capabilities.

3.7.3 Register allocation with advanced processors

These considerations can be extended in a general way to most of the present-day advanced processors. Nowadays, registers are nothing but another level of the memory hierarchy, as they are used to *cache* the most used and stressed variables (memory locations). Registers are loaded and spilled back; data are allocated onto them using complex algorithms that consider number of references and the similar; they store different data throughout program execution. At the hardware level, data cache is usually part of the pipeline, and becomes faster and faster (few cycles are needed to access cached data). On the other hand, register files are growing larger and slower, and in some recent processors they are moved out of the pipeline, and further cached: registers and data caches are getting closer and closer in term of performance and functionalities.

In most of modern commodity processors, registers are the only memory hierarchy level that is managed explicitly by software, with all the problems that this can raise: imprecise knowledge of program's run-time behavior, and thus inefficient data allocation in some situation, complex compile-time algorithm to extract information... It is my opinion that the difference between registers and memory locations should be removed from the ISA, and that the hardware should be responsible for the best register allocation.

For sure, the information collected by compilers should be passed in some form to the hardware, the same way that prefetching instructions can be used to enhance cache performance.

We must also remember that many hardware structures control the data cache, that makes it somewhat a larger register file: advanced processors features logic that controls memory aliasing, memory-carried data dependences and so on. This mechanism may be limited (limited number of entries), but it plays the same role that reference counters and the similar play for the register file.

A new instruction set architecture should be developed, that removes the concept of registers, and leaves the task of data allocation into register to the run-time structures: SSA and dataflow seem good candidates, along with the good ol' memory-memory ISA. To be less radical, off-line optimization (data-reallocation into registers after collecting run-time behavior information) also can be an interesting direction (Smi00).

3.7.4 Cray MTA compiler's result

All the kernels have been run on the *Cray MTA* too. The sequential versions were submitted to the compiler, in order to determine which kind of information could be extracted by its algorithm, and have a term of comparison for my compiling work. Here most interesting results are shortly reported.

Cyclic reduction is used by *MTA compiler* to parallelize kernels no. 5 and 19, in addition to kernel no. 11, previously discussed. Unrolling was used to parallelize partially kernel no. 14 and 18, instead of loop fusion. On the contrary, there are no optimization for kernel no. 2, that I could parallelize with interleaving and loop peeling, kernel no. 17, that features independent iterations, and kernels no. 16, 20 and 24.

Loop interchanging is an opportunity that was not explored, and it is used for kernel no. 1, 7, 8, 9, 10 and 12. It must be said that *Cray MTA compiler* interchanging involved the outmost iteration, that is introduced only to lengthen the execution time, so to have more consistent results, and that was not considered in my rewriting work.

MTA compiler rewrites the accumulation loops, introducing some synchronizations to protect the accumulation variables, and then parallelizing the iterations. The local accumulation in my code can go beyond that, and exploit much more parallelism.

3.8 Overall speedup

As said, good speedup can be reached using different techniques for every single kernel. A description of the global results here obtained is given by figures 3.7 and 3.8, that show two closely related figures: the first plots the average useful IPC, the second the effect of increased number of threads on completion time. The former value is measured as the number of useful instructions that are performed in one cycle, considering synchronization and everything else added to the code as overhead. This is computed by the following:

$$useful\ IPC_i = \frac{sequential\ proc.\ util. * BW}{normalized\ completion\ time\ with\ i\ threads} \tag{3.1}$$

where BW is the maximum sustained IPC, equal to 8.

These two metrics heavily penalize MT versions, as everything added to introduce the MT behavior is seen as an overhead. Nonetheless, this kind of data is the only needed to determine if parallelization is useful. As a matter of fact, for some kernels such as no. 16 and 23, a very high processor utilization does not correspond to any improvement in completion time.

With the techniques here discussed, more than one more instruction per cycle is executed on average (from 2.72 to 3.93), and completion time is reduced of about one fourth (73.91% of sequential version).

As the reader can observe, the best performance is not always reached with eight threads. For accumulation loops, for instance, few threads are usually enough. This raises the question: *how the best number of threads can be determined by the compiler?* This is still an open problem, and a first approach is discussed in section 5.3.1.

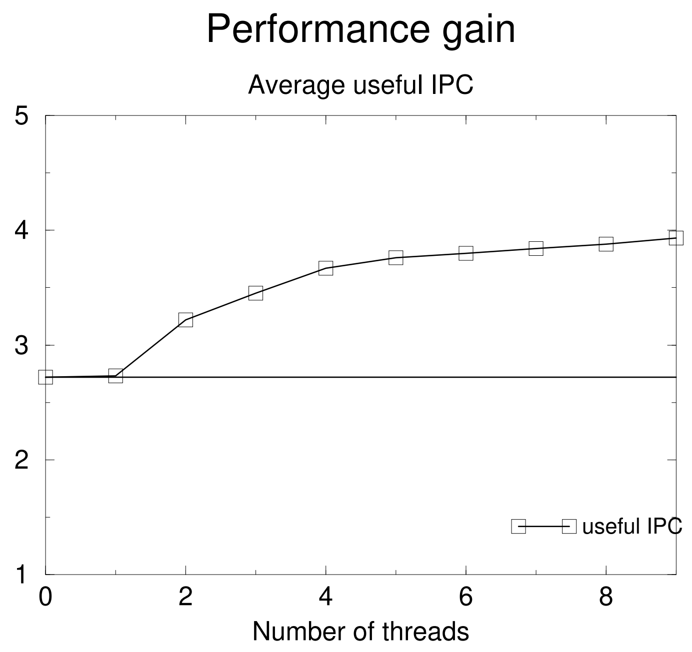


Figure 3.7: Effect of increased number of threads on IPC

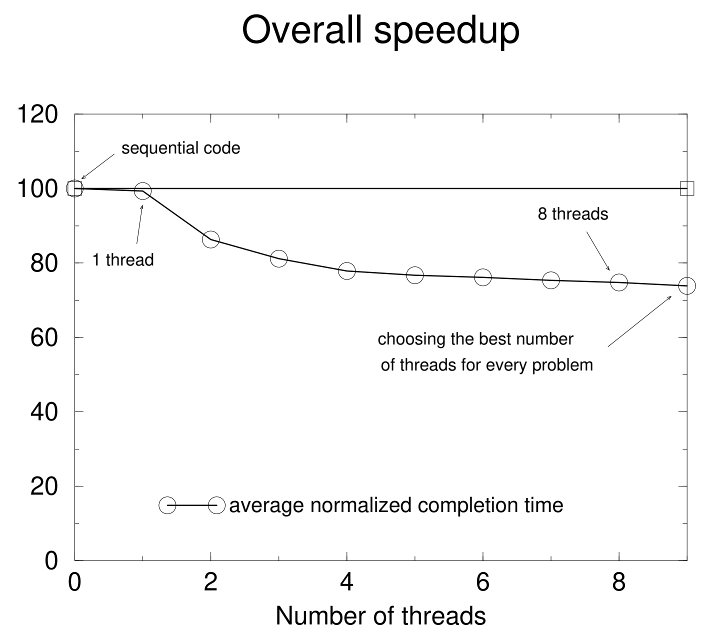


Figure 3.8: Effect of increased number of threads on completion time

Chapter 4

Experimental results

4.1 Independent iterations

Kernel 1 — hydro fragment

This short kernel updates, with a linear scanning, a vector with a very simple function, that depends from values stored in other two vectors. This kernel has a very good processor utilization (66%, that is about 5.2 IPC) with the sequential version, that the MT version can not improve. This is a rather general principle: *high processor utilization limits the opportunity for improvement on SMT*. In this case, the main bottlenek is given, in the MT version, by a small FP instruction queue, that causes a high number of FP queue conflicts (see figure 4.1).

Kernel 7 — equation of state fragment

Similar to kernel no. 1, it features a more complex update function, using three different vectors, and reading 9 elements. A high number of threads here is important to reduce the effects on performance of this keavy group of memory accesses. Performance reached by the eight-thread version is really impressive, more than 94% processor utilization (see figure 4.2).

Kernel 8 — ADI integration

This kernel updates a tree-dimensional vector, reading data from one plan, and writing to the other (with a two-dimensional iteration

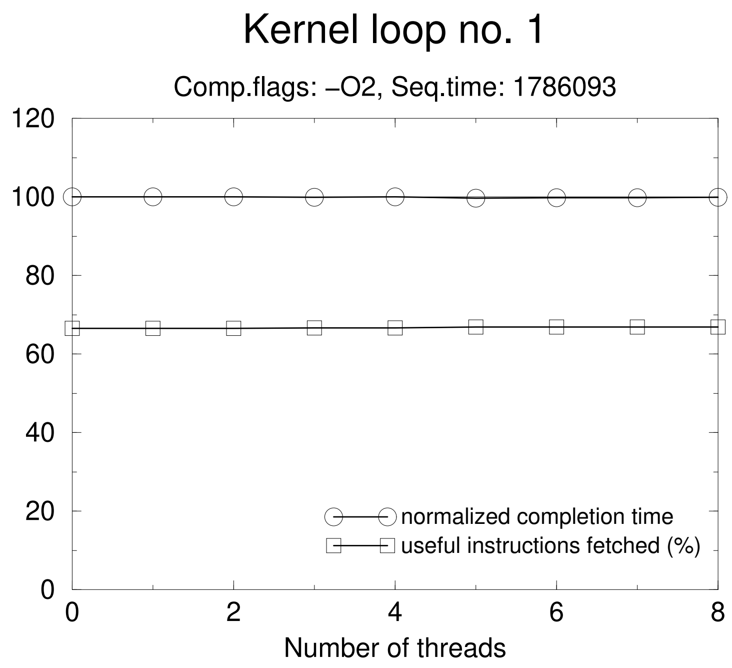


Figure 4.1: Execution statistics for kernel 1

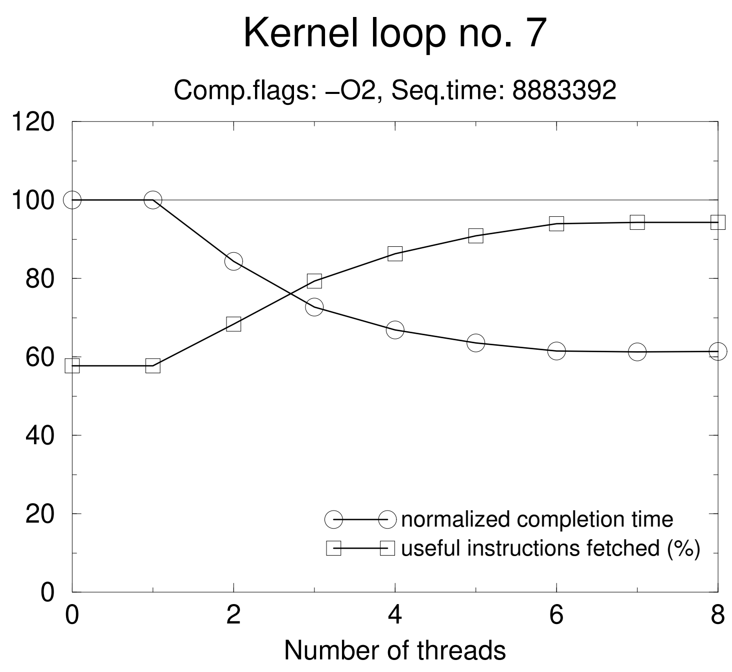


Figure 4.2: Execution statistics for kernel 7

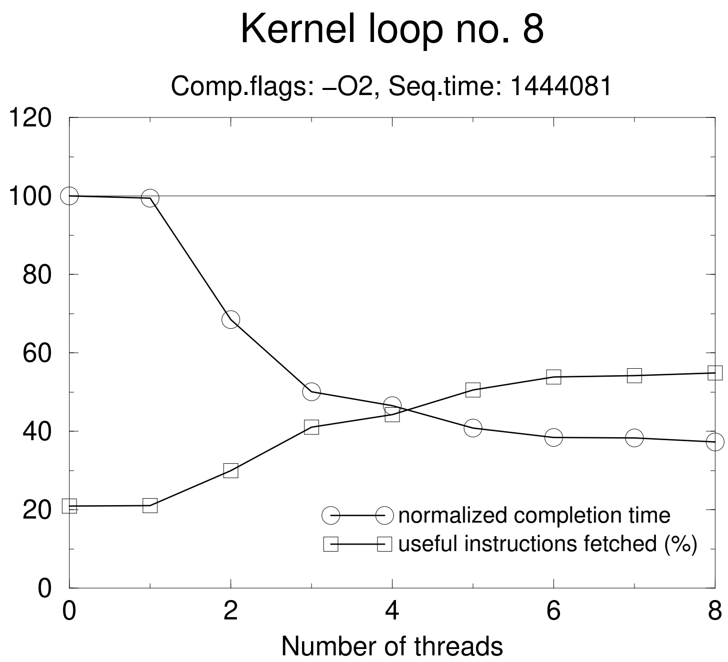


Figure 4.3: Execution statistics for kernel 8

space). To make things even more complex, without any apparent reason, temporary values are stored tidily in one vector, the values of which are not alive outside the specific iteration that creates them. Compiler can surely determine this fact using some simple techniques (comparing indices...): I suppose that it is able to introduce a temporary vector (local to each iteration) to store all the temporary values, and then that it is able to transform access to a three-element vector (as it happens in this case) to access to three distinct variables. This transformation makes iterations independent, as it removes the fake storage-related dependence carried by the temporary vector.

After this change (introduced successfully even in the sequential code), a very good improving could be measured: MT code is more than twice faster (see figure 4.3).

Kernel 9 — integrate predictors

This kernel uses a very large bidimensional vector. For each row, values from all the columns are used to update the element in the

num.th.	seq.	1	2	3	4	5	6	7	8
mem.delay	116.5	116.5	112.8	73.5	59.4	74.0	60.3	51.9	46.2

Table 4.1: Average memory-access delay for kernel no. 9

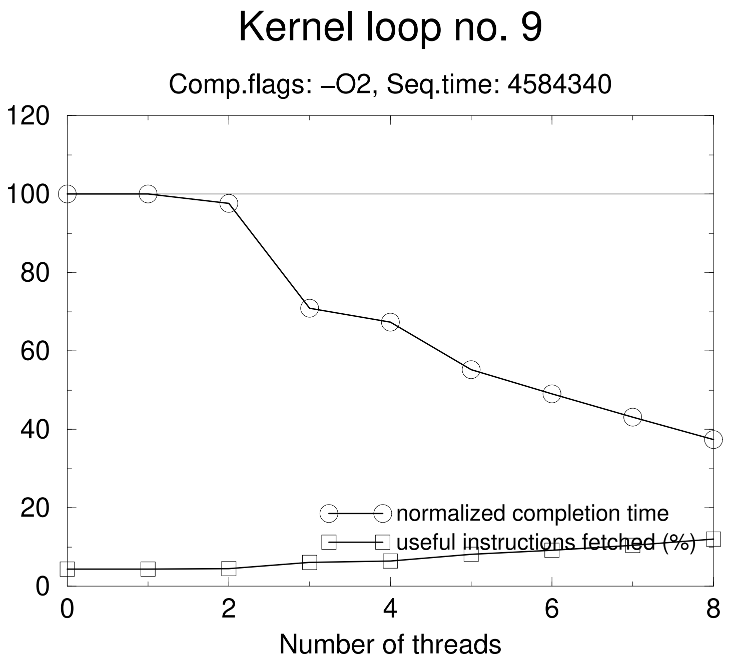


Figure 4.4: Execution statistics for kernel 9

first one, with a linear, loop-independent function. This complex access pattern causes a very bad Dcache utilization (only 70% to 80% Dcache hit ratio), and a very high memory delay (average access time is equal to 116.5 cycles for the sequential version). Due to these problems, processor utilization is really low.

Nonetheless, this makes room for SMT's high latency-hiding capabilities. As it can be seen from table 4.1, memory latency is well hidden by multithreading, and both completion time and processor utilization improve a lot: the eight-thread version is 2.5 times faster than the sequential code. This code shows one of the best improvements recorded in this work (see figure 4.4).

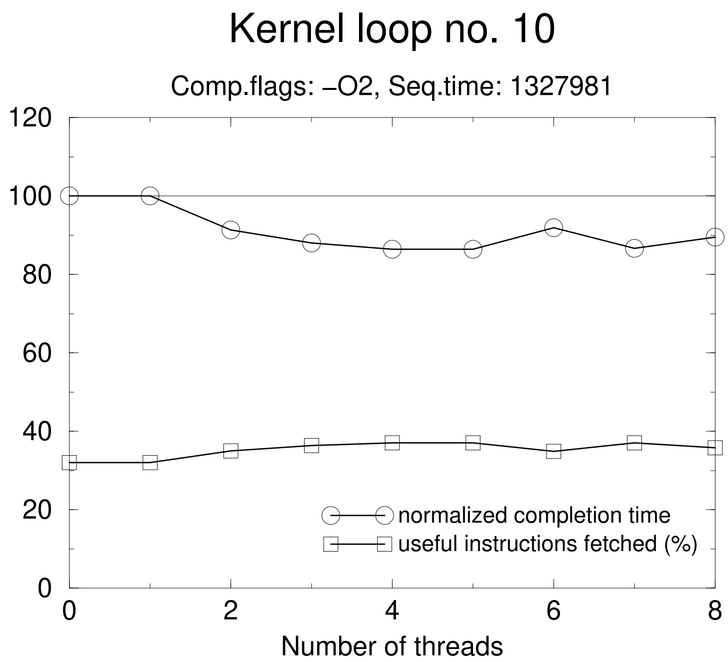


Figure 4.5: Execution statistics for kernel 10

Kernel 10 — difference predictors

This loop features a very heavy loop-independent computation, with a complex and long chain of dependent floating-point operations and memory accesses. Limited improvement is observed (see figure 4.5), due mostly to this complex execution pattern, that features low IPC rate, and to the fact that instruction queues (IQ) and the reordering buffer (ROB) are not large enough to store more copies of the loop-body (see chapter 5).

Kernel 12 — first difference

This loop executes a very simple vector computation (`x[k] = y[k+1] - y[k]`), that improves a lot with the number of threads. Nonetheless, results are really unexpected, as they present a consistent bad-performance spot when three threads are running. Many complex reasons, related to the actual size of some hardware structures, determine this problem, that disappear when other TLB, cache, and IQ sizes are used in the simulator (see figure 4.6).

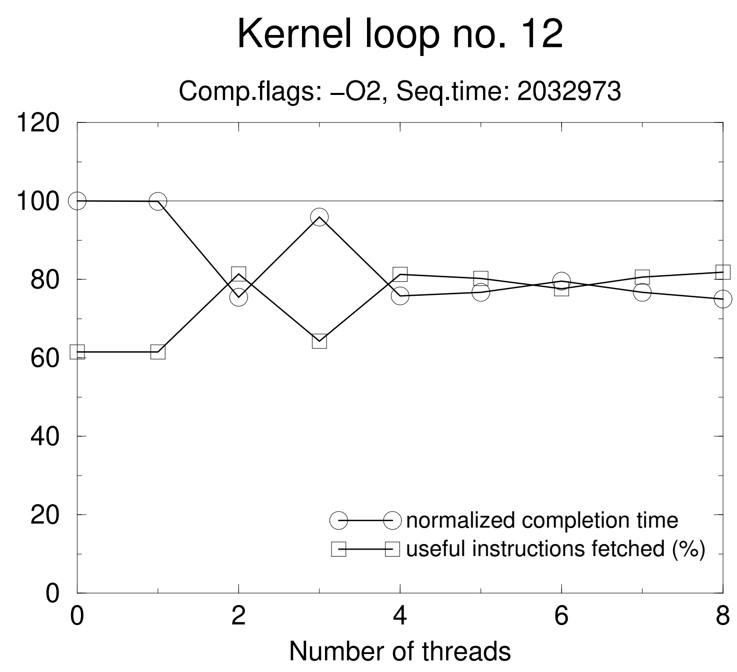


Figure 4.6: Execution statistics for kernel 12

Kernel 15 — casual Fortran

This loop is more complex and long than the previous ones. It features a set of highly irregular nested branches, that depends heavily from data stored in three large bidimensional matrices. Nonetheless, it shows a very good improvement, with simple interleaving (see figure 4.7).

Kernel 17 — implicit, conditional computation

The available implementation of this loop was written *à la Fortran*, with explicit branching (`goto`). As said in the introduction, I have chosen to rewrite this kind of kernels in a more modern, structured way, that allows a better understanding of dependences among instructions and so on. Actually, I am not sure that a compiler can restructure automatically a loop like that, but I suppose that modern programs will be more structured, and the compiler can use this type of information to manage the parallelizing task. Once rewritten in modern C, this code gets really strong improvement (see fig-

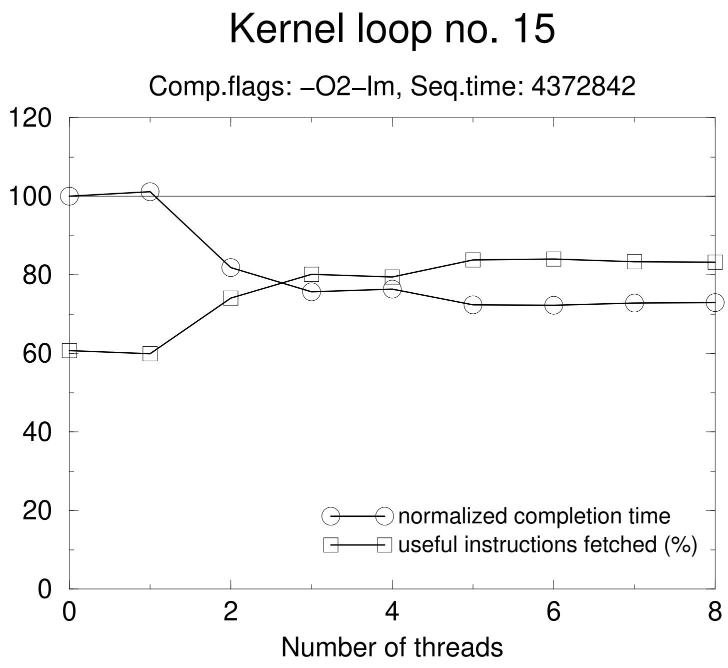


Figure 4.7: Execution statistics for kernel 15

ure 4.8), due to independent iterations, fact that could not be easily recognized in the original version: *Cray MTA compiler*, as a matter of fact, can not introduce any parallelization to this code in its original version.

Kernel 21 — matrix*matrix product

This loop features a very short vector computation, with independent iterations. Simple interleaving offers a small improvement to a really high initial processor utilization (about 80%). The best performance is reached with only three threads, due to increasing conflicts for resources when more threads are used (see figure 4.9).

Kernel 22 — Planckian distribution

This loop computes a very simple update of two vectors, scanning them linearly. Nonetheless, the usage of the `exp` function introduces very high latencies, due to a high number of dependent floating-

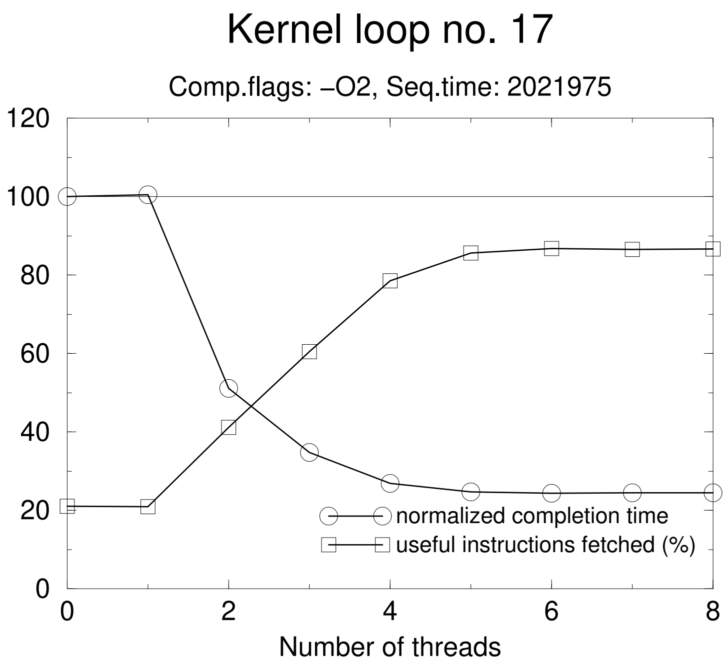


Figure 4.8: Execution statistics for kernel 17

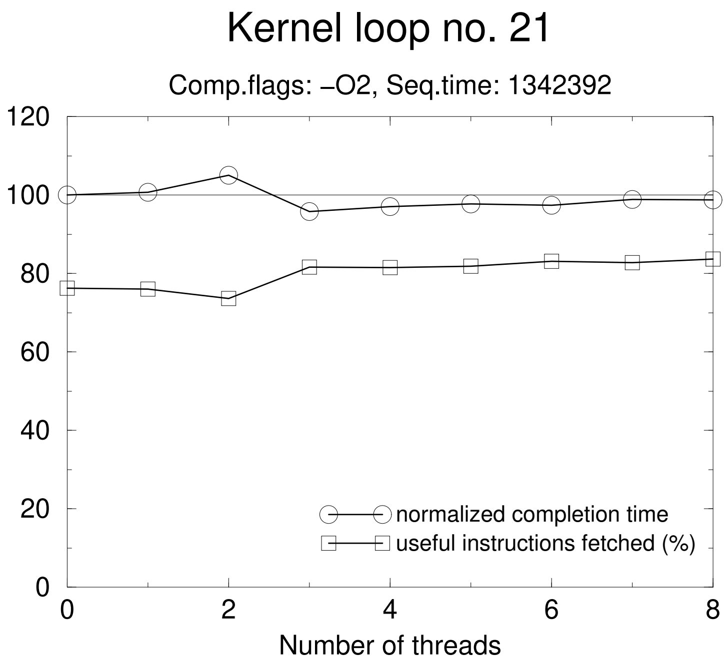


Figure 4.9: Execution statistics for kernel 21

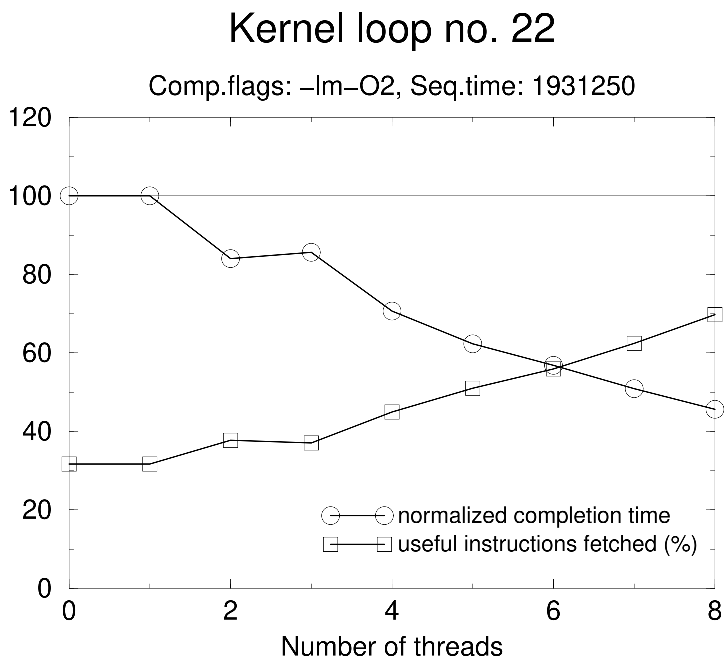


Figure 4.10: Execution statistics for kernel 22

point operations in its computation. In this case, SMT finds a fertile ground, and iteration-interlaving causes really good improvement (see figure 4.10).

4.2 Loop-carried-dependence loops

Kernel 5 — tri-diagonal elimination, below diagonal

This loop computes a very tight recursion (`x[i] = (y[i] - x[i-1]) * z[i]`), that could not be parallelized. The plot shown in figure 4.11 refers to a version in which the update was surrounded by a suitable locking able to keep the right order. This only contributes to a very high overhead, that corresponds to a higher completion time (see figure 4.11).

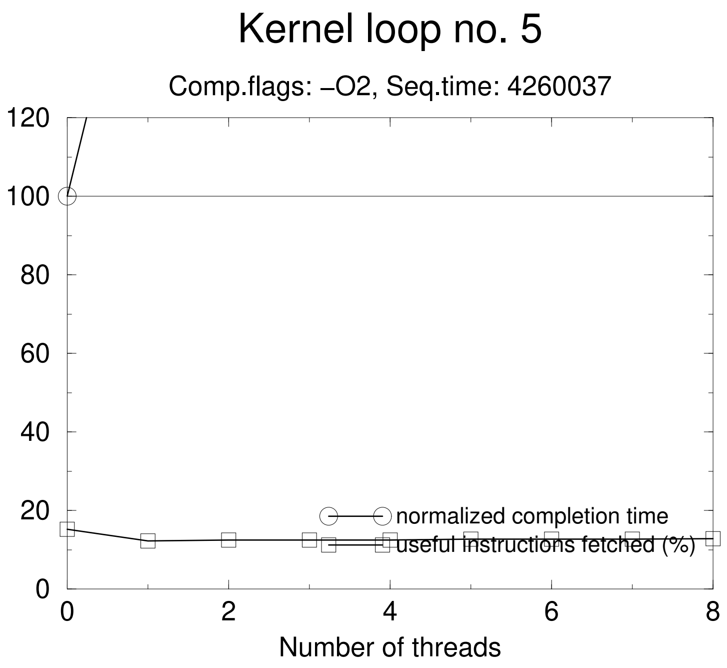


Figure 4.11: Execution statistics for kernel 5

Kernel 11 — first sum

This kernel had a very good improving when parallelized with the cyclic reduction technique. The initial low processor utilization offers large opportunity to introduce advanced techniques. Even if more instructions are executed (about 1.7 times as much), the increase in available TLP allows a much better processor utilization, that can overcome the large overhead. The MT version is almost twice as fast (see figure 4.12).

A full cyclic reduction (recursive binary algorithm) suffered from really high overhead and load unbalance. A simplified version (see program fragment 4.1) was implemented for the SMT.

This algorithm will be rewritten in order to be more general, and will be introduced into a broader rewriting tool for the SMT, that will integrate also the modeling algorithms discussed in chapter 5.

```
#ifdef SINGLE                               //sequential algorithm
    for ( l=1 ; l<=loop ; l++ ) {
        x[0] = y[0];
        for ( k=1 ; k<n ; k++ ) {
            x[k] = x[k-1] + y[k];
        }
    }
#else                                       // MT version
  stride = (SIZE) / (NUMTHREADS);
  if (stride * NUMTHREADS != SIZE) {
      stride ++;  fracstride = 1;
    }
  start = stride * threadid;                // SPMD!! different values
  end = stride * threadid + stride;         // for every thread

  for ( l=1 ; l<=loop ; l++ ) {
    x[start] = y[start];

    for (k = start+1; k< end; k++){         //local running sum
      x[k] = x[k-1] + y[k];
    }
    if (NUMTHREADS == 1) continue;          //done if only 1 thread

    smt_barrier();
    if (!threadid) {                        //running sum of leaders
        for (k=2; k<NUMTHREADS; k++) {
            x[(stride * k)-1] = x[(stride* k) -1] +
                             x[(stride * k) -1 -stride];
          }
        if (!fracstride) {                  //fix last element
            x[SIZE - 1]=x[SIZE-1] + x[SIZE-1 - stride];
          }
      }
    smt_barrier();
    if (threadid) {         //propagating sum
        for (k = start; k< (end-1); k++) {
          x[k] = x[k] + x[start-1];
        }
      }
 }
#endif
```

Program fragment 4.1: Cyclic reduction algorithm for SMT

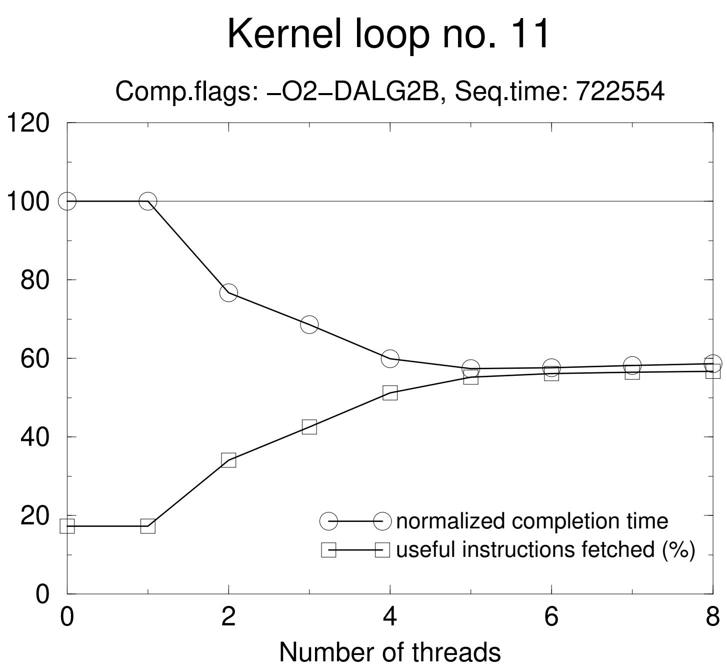


Figure 4.12: Execution statistics for kernel 11

Kernel 19 — general linear recurrence equations

This loop computes two complex recurrences in a row. The same technique of kernel no. 5 (locking around the update) was used. This loop came out not to be parallelizable with straightforward solutions (see figure 4.13).

Kernel 20 — Discrete ordinates transport

This loop features a recurrence carried on by the `xx` vector. Every iteration uses `xx[k]` to evaluate the test guard and `xx[k+1]`. This code results not be parallelizable with standard techniques (see figure 4.14), due to this dependence.

Kernel 23 — 2-D implicit hydrodynamics fragment

As said, an aggressive loop-skewing has been tried for this kernel. It updates a large bidimensional matrix in a simple row-column order, with a function depending from the four neighbors, and from

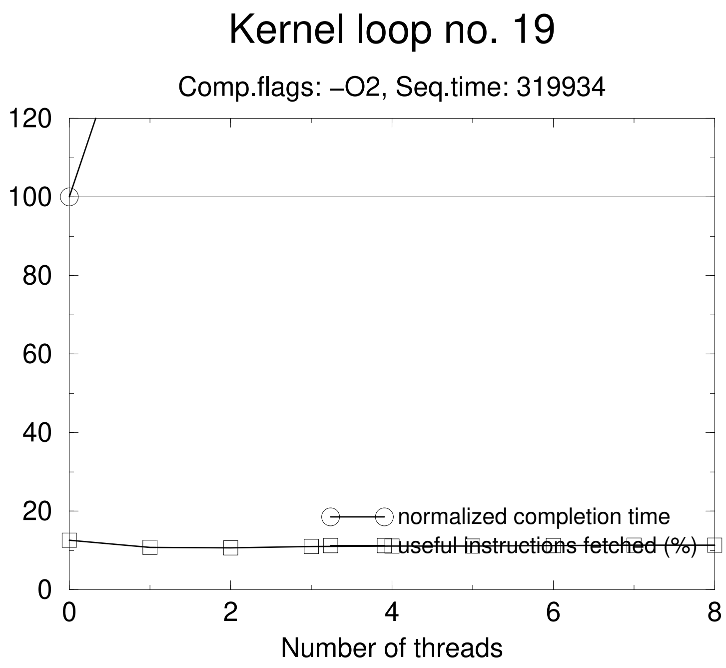


Figure 4.13: Execution statistics for kernel 19

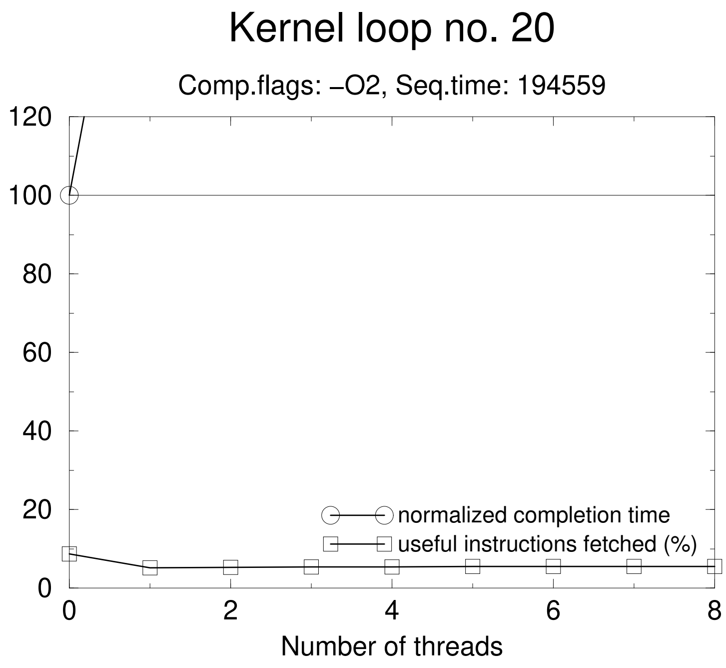


Figure 4.14: Execution statistics for kernel 20

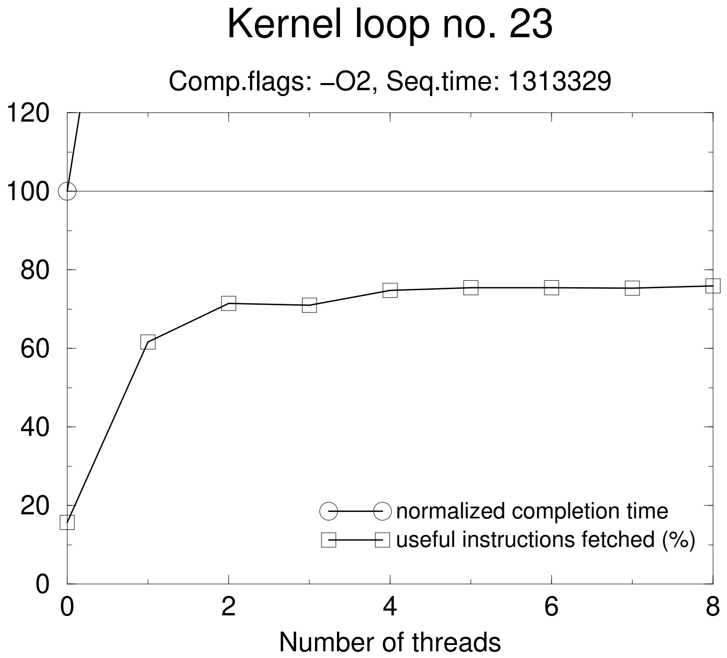


Figure 4.15: Execution statistics for kernel 23

the values of another two matrices, that are only read. Skewing was tried to exploit the so-called *tidal-wave* parallelism, but the implemented algorithm had a too high overhead to be useful (see figure 4.15): the very high processor utilization is not followed by any speedup. Nonetheless, I expect that a careful tuning can improve performance. Tiling also could be used to exploit parallelism.

4.3 Accumulation loops

Kernel 3 — inner product

This loops accumulate the values of `z[k] * x[k]` in a global floating-point variable. It was parallelized, introducing, in every thread, a local variable that sums all the values computed by it, and then accumulating local sums (*local accumulation*)[1]. Good improvement in performance (see figure 4.16), increasing till the 4-thread version.

[1] I did not consider any problem of numerical stability: the MT version does not guarantee the order of the summation.

(a) Original code

```
for ( l=1 ; l<=loop ; l++ ) {
    q = 0.0;
    for ( k=0 ; k<n ; k++ ) {
        q += z[k]*x[k];
    }
}
```

(b) Modified code

```
q= 0.0;
for ( l=1 ; l<=loop ; l++ ) {
    temp = 0.0;
    for ( k=0 ; k<n ; k += 1 ) {
        temp += z[k]*x[k];
    }
    q += temp;
}
```

Program fragment 4.2: Which one is the fastest one? The second one!

An interesting note: the introduction of the local variable had a very good effect even for the 1-thread version. This change was applied so to sequential code, that reported a good improvement (completion time was 40% better!). It is an open problem whether a smart compiler (smarter than *gcc*) can introduce automatically this kind of optimization (see program fragment 4.2). This phenomenon was recorder also for kernel no. 6 (below).

Kernel 4 — banded linear equations

This loop computes three accumulations, that are stored into three distinct element of a large vector, that are the only global variables updated. Local sums could be carried on independently (see anyway note 1 on the preceding page). Good improvements with the second thread. As explained in section 3.5, the ordering introduced by locking makes further threads useless (see figure 4.17).

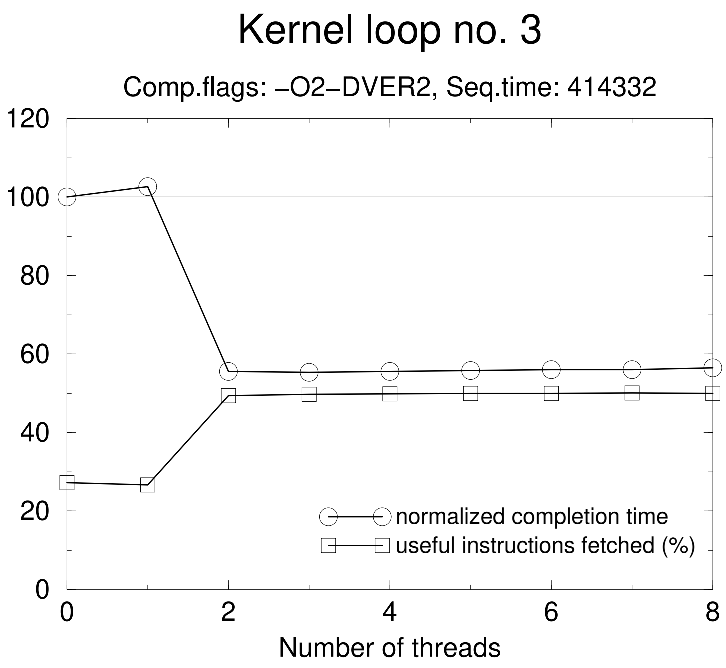


Figure 4.16: Execution statistics for kernel 3

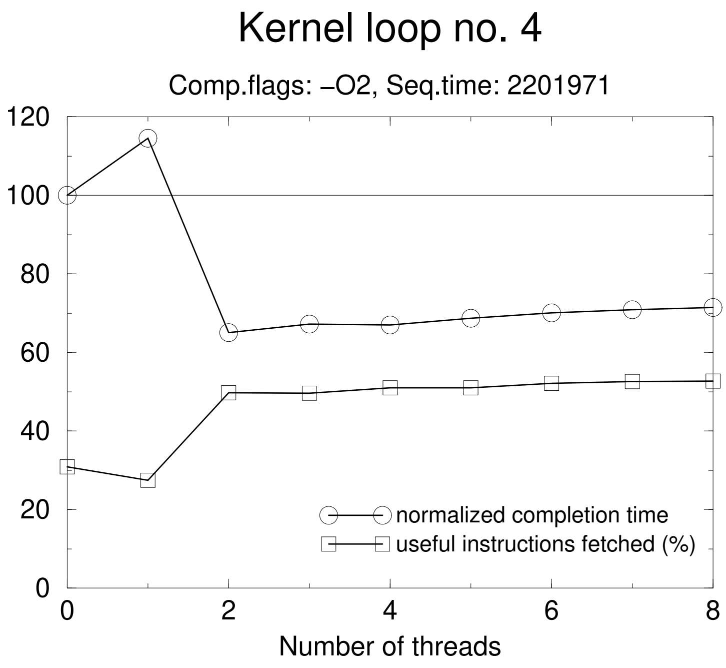


Figure 4.17: Execution statistics for kernel 4

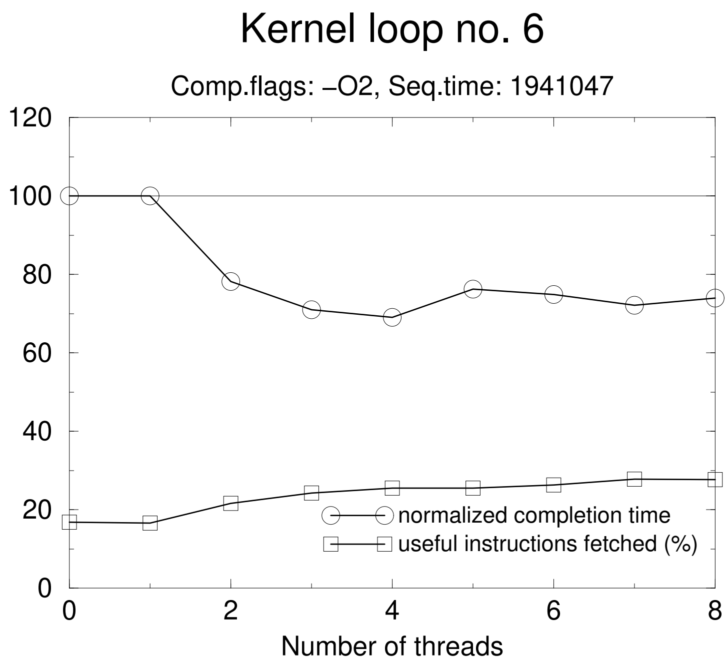


Figure 4.18: Execution statistics for kernel 6

Kernel 6 — general linear recurrence equations

This kernel perform a particular triangular recurrence (`w[i] += b[k][i] * w[(i-k)-1]`), led by a double nested loop. It could be parallelized as the updated vector element stores a long accumulation, that is used only by the next outer iteration. The accumulation variable was made local to each thread, and local results were accumulated in the right order, through ordering locking (this solution for kernel 6 is discussed in (TLEL99), where some consideration about the best parallelization can be found). Few threads are enough to reach the best performance (see figure 4.18).

As said for kernel no. 3, optimization introduced to have a better parallelization are useful to the sequential version, too.

Kernel 13 — 2-D PIC (Particle In Cell)

This kernel executes a complex set of data-dependent memory-accesses, and features an accumulation that causes a possible loop-carried data dependence: an element of the large bidimen-

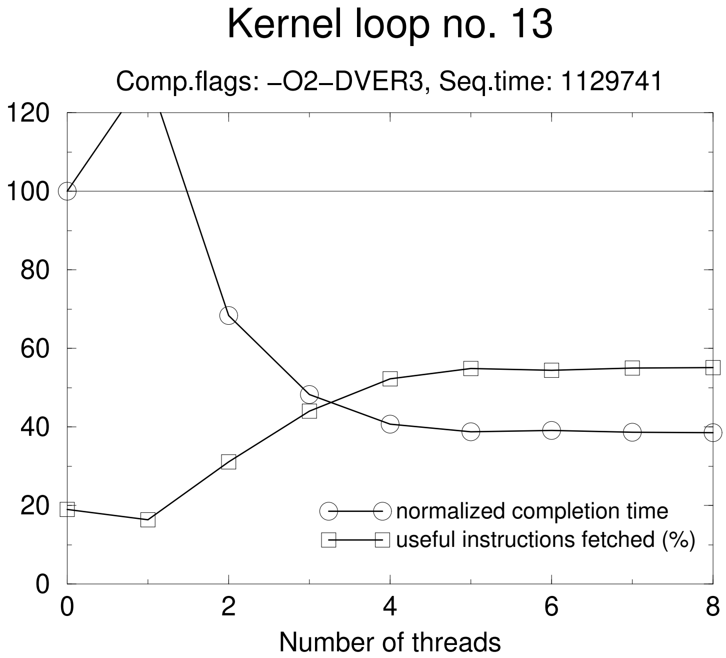


Figure 4.19: Execution statistics for kernel 13

sional `h` matrix, determined in a complex data-dependent way, is used to accumulate. Three versions of this loop have been tested: the first one used one single lock to control access to the whole `h` vector, the second one used one lock per column, the third one used a bidimensional lock matrix, with one lock for each element of `h`. Even if this last solution has a large overhead due to lock initialization, speedup is the best of all, and comes out to be very good: 2.5 times faster than sequential code (see figure 4.19). As said, *a high number of threads can be useful to overcome a large overhead*.

4.4 Larger loops

Kernel 2 – ICCG excerpt (Incomplete Cholesky Conjugate Gradient)

This loop computes a complex recurrence, in which last iteration depends from values computed in the first one, while the other ones are independent one from another, and their results are used only in

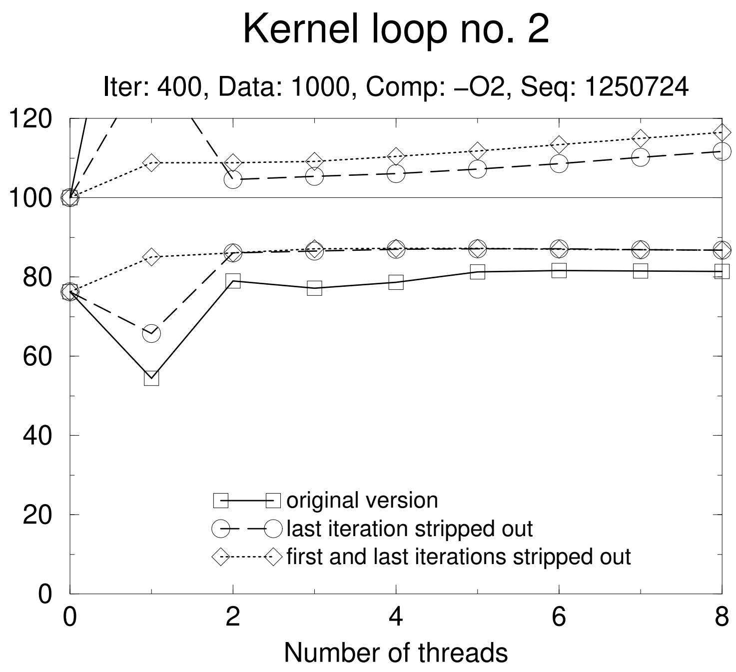


Figure 4.20: Execution statistics for kernel 2

the next outer iteration. The basic approach (interleaving + locking) improved when the first and the last iteration were peeled out of the loop-body: this type of optimization can be successfully completed by a smart compiler. Nonetheless, overhead for this solution (some barriers and locks are needed to guarantee the correct order) was too high to allow any performance improvement with respect to sequential code, even if parallelization overhead was almost completely absorbed in the two-thread version, that proved to be the best. This is another interesting case in which the determination of the best number of threads is particularly critical and difficult.

This code comes out not to be successfully parallelizable with standard techniques. In figure 4.20, a comparison for different implementations can be found.

Kernel 14 — 1-D PIC (Particle In Cell)

This kernel describes a very complex computation, carried on by three small loops, the iterations of which are independent. As discussed in (TLEL99), where this kernel is also studied, the three loops

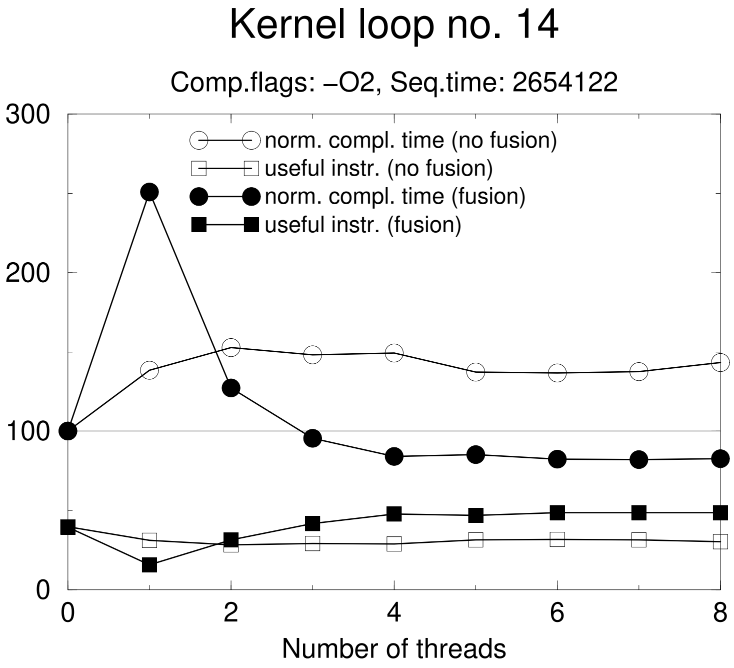


Figure 4.21: Execution statistics for kernel 14

are just a smart splitting of a larger loop, to increase ILP. When loop are fused, higher opportunity for parallelism emerges, that is exploited when more threads are used. This solution is not good with few threads, as loop distribution offers a really high ILP, that is well exploited by large SMT's superscalar bandwidth.

In figure 4.21, a comparison between the two versions (loop fusion and loop distribution) is given. From the point of view of the compiler, it means that *more versions* of the same program have to be available at run-time, that can be chosen accordingly to the number of threads available to the program (in a multiprogrammed environment). This idea is gaining importance even for traditional processors, under the broader denomination of *feedback-directed optimization* (Smi00).

Kernel 16 — Monte Carlo search loop

The semantics of this piece of code is not clear: loop-body is composed by a complex chain of nested branches. A dataflow analysis of it shows that only the integer variable `k3`, that is incremented only

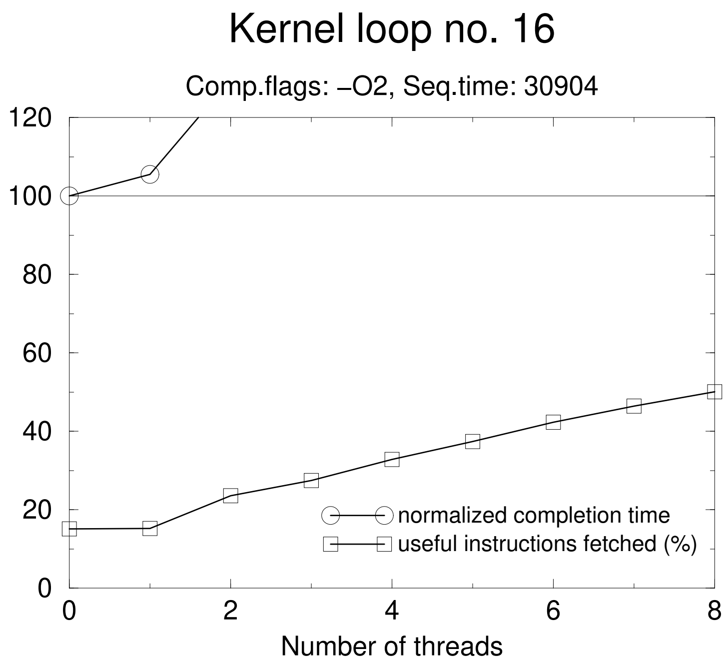


Figure 4.22: Execution statistics for kernel 16

in one of the branches, can be the cause of a loop-carried dependence. It was surrounded by a simple lock, as it is an integer variable, incremented by a constant value, so the order of update is not important. For the rest, iterations are independent, and were interleaved. This technique is not successful with this kernel (see figure 4.22): good parallelization needs probably to be guided by the programmer, but, as explained in the introduction, main goal of this work was to determine what a compiler can do in an automatic way, with general techniques.

Kernel 18 — 2-D explicit hydrodynamics fragment

This kernel is somewhat similar to kernel no. 14: the loop-body is split into three shorter loops, that could be successfully fused. Loop fusion shows good performance improvement, even if with few threads loop distribution proves to be better (see figure 4.23). This is another interesting case in which the determination of the best implementation is particularly critical, and dependent from the number of available contexts.

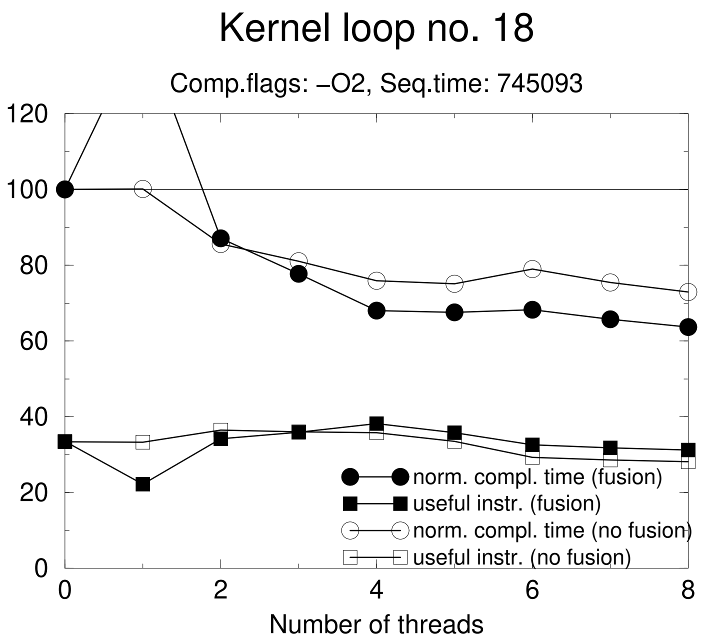


Figure 4.23: Execution statistics for kernel 18

Kernel 24 — find location of first minimum in array

This loops scans a vector looking for the first minimum. It was parallelized successfully (see figure 4.24) considering the minimum as an accumulation, using local variables to store the local minimum for each thread. This type of restructuring can be debatable, as this requires the compiler to recognize that the test `x[k] < x[m]` is a way to compute an accumulation function, but I believe that the usage of a library function `min` could make this transformation automatic (some libraries as *MPI* features specific parallel implementation for the minimum). These changes proved to be successful.

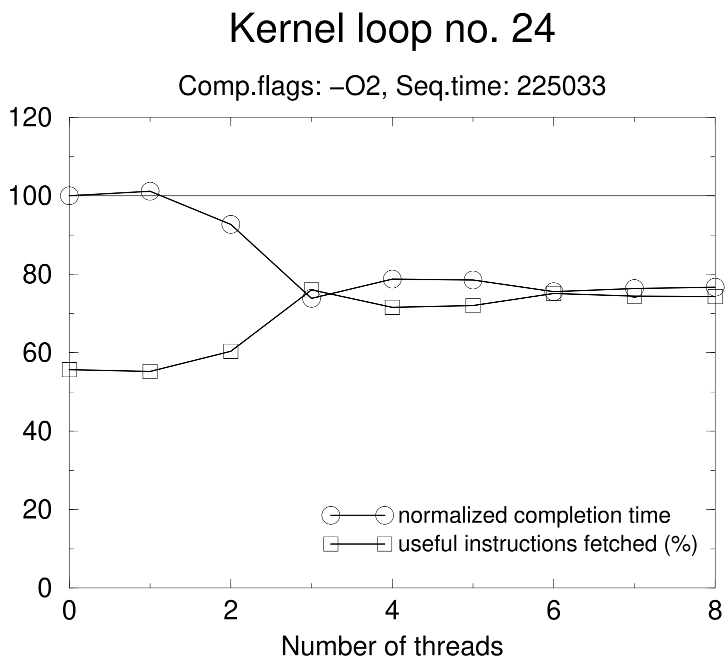


Figure 4.24: Execution statistics for kernel 24

Chapter 5

Modeling SMT performance

In this section, a model of performance for programs running on the SMT processor is presented. It can predict performance for sequential and multithreaded code using static information as size of basic blocks and critical path length. The idea here discussed can be extended and adapted to other superscalar processors.

5.1 Methodology

Extensive use of the *Atom* tool (Com), offered as a part of the *Alpha C compiler* libraries, has been done. *Atom* offers the user the opportunity to augment the assembler code with detailed (instruction-level) routines for instrumentation and analysis. I developed two simple tools, with which I collected the experimental data here discussed.

The first, *basicblox*, is a very detailed profiler, that is able to determine which basic blocks are most used during actual execution. It was vary useful to limit further analysis to smaller parts of the code. Actually, the information gathered showed that the most stressed basic blocks, in many Livermore loops (14 out of 24), are executed for more than 98% of the time: limiting analysis to those blocks does not have a great impact on accuracy.

Atom is able to instrument the code at the entrance of each basic block. A simple counter of the number of times each block is executed, and of the total number of instructions it executes, could be simply implemented. With this information, it is straightforward to determine which the most stressed blocks are.

The second, *CPL*, tries to extract, for the most loaded blocks (as suggested by *basicblox*), the length of the *critical path* (CPL). The critical path is defined as *the longest (in term of latencies) subset of instructions from the block, linked by data dependencies*. The algorithm is very simple, but results are really accurate. As a matter of fact, code compiled from *gcc*, the GNU version of C compiler (the compiler used throughout all the simulation, as explained in section 3.2), is very simple and predictable. (The model was tested also with code produced by *cc*, the Digital version of the C compiler, but results are actually worse, as described in section 5.2.3.)

I observed that most of load operations (that set registers' value) are in the beginning of blocks, so we can consider them as independent, and with no precedence to wait for. The algorithm computes, for every instruction, the number of cycles, from the beginning of the block, it has to wait before all the values needed are ready. Let's call this value $dependence\ delay$. It is computed as:

$$dep.\ delay_x = \max_{\text{all } y\text{: } x \text{ depends from } y} \left(dep.\ delay_y + latency_y\right) \tag{5.1}$$

The algorithm has a very simple model of the memory hierarchy, in which load instructions have latency equal to the average memory access time[1]. In the present implementation, this value is collected by a short profiling of the code (SMTSIM reports the average memory-access delay at the end of the simulation).

The maximum of the $dependence\_delay$ over all the instructions in the block represents the minimum number of cycles that a single copy of the block can take. Let's call it $critical\ path\ length$ or CPL.

Furthermore, the implemented algorithm heuristically recognizes instructions as loop-counter increments or base-address updating, that appear in the code as independent from every other instruction in the block, but dependent from itself. For an instruction x recognized as loop-critical, i.e. that depends from values computed in the previous iteration, let's call $loop\ dependence\ delay$ the time that is needed to perform it. A value coming from the previous iteration will not be available before this time. If another instruction y depends from x, y's $loop\ dependence\ delay$ is equal to x's $loop\ dependence\ delay$, plus y's $latency$.

[1]More exactly, the latency is equal to the maximum of 2, i.e. the pipeline latency of a load instruction, and the average memory access time.

Assembler code	depends from	dep. delay	loop dep. delay
0 (0x20005960) : ldt \$f1, 0(t1)	-	0	0
1 (0x20005964) : ldt \$f10, 0(t2)	-	0	0
2 (0x20005968) : subt \$f1,\$f10,\$f1	0,1	2	0
3 (0x2000596c) : addq t2, 0x8, t2	3	0	1
4 (0x20005970) : addq t4, 0x1, t4	4	0	1
5 (0x20005974) : addq t1, 0x8, t1	5	0	1
6 (0x20005978) : cmplt t4, a1, t0	4	1	1
7 (0x2000597c) : stt \$f1, 0(t3)	2	6	0
8 (0x20005980) : addq t3, 0x8, t3	8	0	1
9 (0x20005984) : bne t0, 0x20005960	6	1	2
		CPL:	6
		loopCPL:	2

Table 5.1: Example of CPL's processing, kernel no. 12

$$loop\ dep.\ delay_x = \max_{\text{all } y:\ x \text{ depends from } y} (loop\ dep.\ delay_y + latency_y) \quad (5.2)$$

For independent-iteration loops, the $loopCPL$ is the maximum of loop dependence delays, over all instructions of the block. In the other cases, this parameter was set manually to the correct value, as it could be measured in the disassembled code, compared with the source code: the implemented algorithm is not able to recognize dependences that are carried by memory.

In table 5.1, an example of *CPL*'s processing can be found. The latency values described in table 3.2 can be used for comparison.

5.2 Mathematical model of SMT performance

5.2.1 Role of loop-carried critical path

The number of integer, load/store and floating-point instructions for every block is counted at compile-time by *basicblox*. These data are compared against the number of functional units available to determine an upper limit for performance. Another key factor in determining this figure is the size of reordering buffer (ROB) and instruction queues: a larger ROB and longer instruction queues allow the processor to manage instructions from more iterations, and use them to hide latencies of one iteration with instructions from the other ones. I would like to recall the fact that processors with advanced out-of-order execution capabilities (as SMT) can behave somewhat as a dataflow architectures, allowing the execution of instructions as soon as data are available, even forcing the sequential order: if loop-counter updating is fast, more copies of the loop-body can be present at the same time in the processor.

Suppose that enough copies of the loop-body (in case of independent iterations) are together active (queued) in the processor, so all the latencies that participate in the critical path can be completely hidden. In this case, a single iteration will take no less than $loopCPL$ cycles (CPI: cycles per iteration), that determines an upper bound on performance.

$$ideal\ best\ CPI = loopCPL \tag{5.3}$$

$$ideal\ best\ proc.\ util. = \frac{size}{loopCPL * BW} \tag{5.4}$$

Here $size$ is the size (number of instructions) of the basic block, and BW the maximum sustained execution bandwidth, that in this simulation is limited by the fetch bandwidth to 8 instructions per cycle. Note that $ideal\ best\ proc.\ util.$ can be greater than 100%, if the loop-carried critical path is short.

Now, let's consider, for every instruction type, how many functional units are availble and for how many cycles. Recalling data presented in table 3.3, the simulated SMT can execute, in one cycle, up to 6 integer operations, of which up to 4 memory operations and 2 synchronization operations, and up to 3 floating-point operations. To get full performance, loop instructions have to fit the available number of slots. We have a degradation if one of the following holds:

$$\#integer\ instructions > loopCPL * 6 \tag{5.5}$$
$$\#memory\ instructions > loopCPL * 4 \tag{5.6}$$
$$\#synchronization\ instructions > loopCPL * 2 \tag{5.7}$$
$$\#floating\ point\ instructions > loopCPL * 3 \tag{5.8}$$
$$size = \#all\ instructions > loopCPL * 8 \tag{5.9}$$

In these cases, more cycles are needed to execute all the operations. The last condition is determined by the limited fetch bandwidth (there are 9 functional units, but only 8 instructions can be fetched per cycle).

Let's consider the minimum number of cycles that is needed to complete a loop iteration, if one of the previous conditions holds. Let $number_i$ be the number of instructions of every type (integer, memory, synchronization, floating-point, and all of them together) and $bandwidth_i$ the maximum number of them that can be executed per cycle (respectively, 6, 4, 2, 3, 8). Let the $saturation\ factor$ be a description of how much functional units are stressed, as follows:

$$saturation\ factor = \max\left(1, \max_{1..5}\left(\frac{number_i}{loopCPL * bandwidth_i}\right)\right) \tag{5.10}$$

$saturation\ factor * loopCPL$ is the minimum number of cycles per iteration, as limited by execution bandwidth ($saturation\ factor \geq 1$). A new (more precise) upper limit on performance is so:

$$best\ CPI_{loop} = loopCPL * saturation\ factor \tag{5.11}$$
$$best\ proc.\ util._{loop} = \frac{size}{loopCPL * saturation\ factor * BW} \tag{5.12}$$

In case of loop-carried dependences, of the type `x[i] = f( x[i-1] )`, this is the best result we can expect, as iterations can not overlap. I want to recall that in this case, anyway, the actual value of $loopCPL$ has to be computed by hand, as the implemented heuristics expects iterations to be independent, and can not track memory-carried dependences. Again, in case of loop-carried dependence, the value reported as CPL in not significant, and the main bound to performance is given by $loopCPL$.

5.2.2 Role of loop-independent critical path

For loops featuring independent iterations, there are other limiting factors. As said, these performance figures can be reached if enough copies of the loop can interleave in the pipeline. Main limit is given by ROB's size: ROB stores instructions in-order, i.e. it keeps a consecutive subset of computation. Instructions can be chosen and moved to instruction queues out of order, and so instruction queues are less a limiting factor. An estimation of reordering structures' latencies-hiding capabilities, comes counting how many copies can be stored in the same time in the reorder buffer. Nonetheless, the simulator I could use had only a basic model describing the reorder buffer. Thus, its size will be approximate by integer and floating-point instruction queues' size that, in the simulated model, are 32-instruction long each. The number of fully stored copies will be:

$$copies = \min\left(\frac{\#integer\ instructions}{32}, \frac{\#floating-point\ instructions}{32}\right) \tag{5.13}$$

(Here, memory and synchronization instructions are counted with integer instructions, as they share the same queue.) Consider the following performance estimation:

$$best\ CPI = CPL \tag{5.14}$$

$$best\ proc.\ util. = \frac{size}{CPL * BW} \tag{5.15}$$

We can expect a performance improvement, with respect to these values, of about $copies$ times, as the execution of $copies$ copies is interleaved in the processor: among different threads there are no dependences, and if the execution bandwidth is large enough, more copies will be executed in the time initially took by only one. Obviously, superscalar bandwidth will be a limiting factor, as the replicated copies will be competing for functional units. The $saturation\ factor$ needs to be recomputed as:

$$saturation\ factor = \max\left(1, \max_{1..5}\left(\frac{number_i * copies}{CPL * bandwidth_i}\right)\right) \tag{5.16}$$

So we expect that performance will be no better than:

$$best\ CPI = \frac{CPL * saturation\ factor}{copies} \tag{5.17}$$

$$best\ proc.\ util. = \frac{size * copies}{CPL * saturation\ factor * BW} \tag{5.18}$$

But this is not the case. As a matter of fact, in many loops, CPL is determined mostly by floating point operations, even if the number of copies is limited by a high number of integer operations (memory operations, loop-counter increments...). It can be said that, as integer operations are faster, integer instruction queue is not the problem, and actually the maximum should be used in equation 5.13. Both minimum and maximum number of copies are important to determine the performance: they give a lower and an upper bound on processor utilization. The fact that actual performance will be closer to one of these two values rather than to the other depends from the degree of interdependence between integer and floating-point operations.

So, we have:

$$min.\ copies = \min\left(\frac{\#int.instr.}{32}, \frac{\#FPinstr.}{32}\right) \tag{5.19}$$

$$max.\ copies = \max\left(\frac{\#int.instr.}{32}, \frac{\#FPinstr.}{32}\right) \tag{5.20}$$

$$worst\ CPI_{independent} = \frac{CPL * saturation\ factor_{min}}{min.copies} \tag{5.21}$$

$$worst\ proc.\ util._{independent} = \frac{size * min.copies}{CPL * saturation\ factor_{min} * BW} \tag{5.22}$$

$$best\ CPI_{independent} = \frac{CPL * saturation\ factor_{max}}{max.copies} \tag{5.23}$$

$$best\ proc.\ util._{independent} = \frac{size * max.copies}{CPL * saturation\ factor_{max} * BW} \tag{5.24}$$

where $saturation\ factor_{min}$ and $saturation\ factor_{max}$ are computed using the value of $min.\ copies$ and $max.\ copies$ in (5.16).

For independent-iteration loops, the limits described by 5.11 and 5.12 also holds, so we have:

$$best\ CPI = \min\left(best\ CPI_{loop}, best\ CPI_{independent}\right) \tag{5.25}$$

$$best\ proc.\ util. = \min\left(best\ proc.\ util._{loop}, best\ proc.\ util._{independent}\right) \tag{5.26}$$

It can be the case where:

$$worst\ proc.\ util._{independent} > best\ proc.\ util._{loop} \tag{5.27}$$

In this situation, $best\ proc.\ util._{loop}$ will be used also as the lower performance limit (along with $best\ CPI_{loop}$).

For loops with loop-carried dependences, instead:

$$best\ CPI = best\ CPI_{loop} \tag{5.28}$$

$$best\ proc.\ util. = best\ proc.\ util._{loop} \tag{5.29}$$

as we expect no improvements from interleaving.

5.2.3 Experimental results with Atom

The model described above can estimate maximum and minimum performance for loops consisting of one block. It will be tested with the kernels in which just one basic block is executed most (more than 98%) of the time. This allows to compare the estimation with the overall performance of the programs, without great loss of accuracy.

There are many examples of these simple loops in the studied benchmark:

- independent iterations:
  - kernel 1 — hydro fragment,
  - kernel 7 — equation of state fragment,
  - kernel 8 — ADI integration,
  - kernel 9 — integrate predictors,
  - kernel 10 — difference predictors,
  - kernel 12 — first difference,
  - kernel 21 — matrix*matrix product,
- loop-carried-dependence loops,
  - kernel 5 — tri-diagonal elimination, below diagonal,
  - kernel 11 — first sum,

- kernel 23 — 2-D implicit hydrodynamics fragment,

- accumulation loops,
  - kernel 3 — inner product,
  - kernel 4 — banded linear equations,
  - kernel 6 — general linear recurrence equations.
  - kernel 13 — 2-D PIC (Particle In Cell),

In table 5.2 and figure 5.1, experimental data and results of modeling are shown[2]. Please, note that accumulation loops are here described as independent-iteration loops: as a matter of fact, just few instructions (the actual accumulation) contribute to $loopCPL$, while the main computation can be considered part of CPL. Once computed these two figures by hand (the implemented heuristics cannot manage well this type of loops), the model fits easily.

As it can be seen, most of the studied kernels perform closely to the expected maximum. This correspond to the maximum number of copies as in equation 5.20. Kernel no. 13 is closer to the minimum as most of array indices are data-dependent (critical path is composed by a chain of mixed integer and floating-point instructions), so the two instruction queues move with the same speed. A more important problem is presented by kernel no. 7, and in smaller terms by kernel no. 8: they perform quite better than what the model can predict. They feature basic blocks larger than the average: the expected number of $copies$ is smaller than 1, and the expected performance is consequently reduced, even if there is no real slowdown (number of copies is important when computing speedup). I think that another problem offered by their size is that smaller parts of the loop-body (not whole copies) can overlap to hide latencies, and the model here presented cannot determine the measure of this effect.

The described model reaches very good results with short kernels. With loops shorter than 40 instructions, predicted values are not farther than 5% from the maximum (average distance being 6.4%). It is also really good with loop-carried-dependence loops: on average, expected performance is not farther than 1% from actual value (3% in the worst case).

[2] In the figure, the shaded area correspond to the interval between the minimum and the maximum expected performance; the solid line describes actual performance

kernel no.	type	size	CPL	loopCPL	actual performance	best proc. util. loop	worst proc. util. ind.	best proc. util. ind.	expected MIN	expected MAX
1	IND	19	21	2	66.52	100.00	25.85	72.38	25.85	72.38
3	IND	9	6	4	27.06	28.12	85.71	96.43	28.12	28.12
4	IND	9	6	4	29.81	28.12	85.71	96.43	28.12	28.12
5	LOOP	13		10	15.20	16.25			16.25	16.25
6	IND	13	32	8	16.82	20.31	14.77	81.25	14.77	20.31
7	IND	41	34	2	57.75	96.09	19.29	30.15	19.29	30.15
8	IND	122	84	2	21.09	100.00	7.00	14.90	7.00	14.90
9	IND	32	200	2	4.33	70.59	3.76	4.27	3.76	4.27
10	IND	34	46	2	32.69	85.00	11.83	32.85	11.83	32.85
11	LOOP	10		7	17.41	17.86			17.86	17.86
12	IND	10	6	2	61.52	62.50	74.07	83.33	62.50	62.50
13	IND	86	38	6	19.48	93.48	13.12	53.25	13.12	53.25
21	IND	13	11	2	76.68	81.25	42.98	88.64	42.98	81.25
23	LOOP	33		34	15.71	12.13			12.13	12.13

Table 5.2: Experimental results, with gcc code

Model of performance

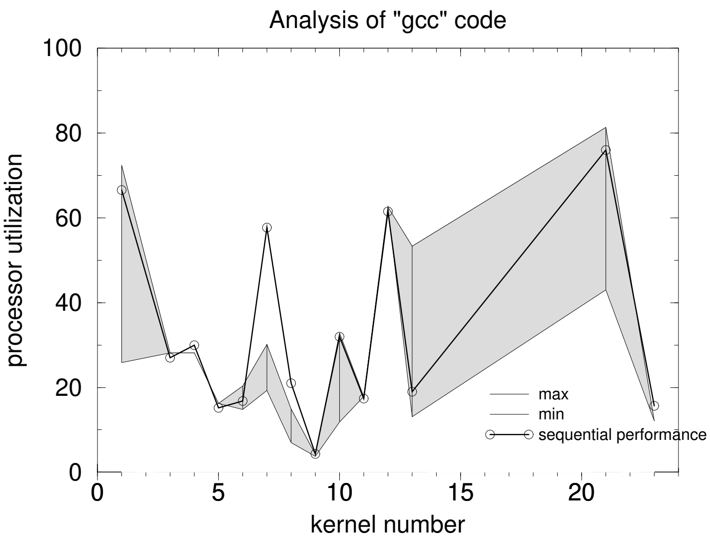


Figure 5.1: Results of modeling performance with gcc code

In table 5.3 and figure 5.2, the reader can find how the modeling methodology works with *cc* code. *cc* compiler is much smarter than *gcc*, and its code is less straightforward to understand and model. The simple algorithm of *CPL* does not offer good results with it, that affects dramatically the accuracy of modeling.

5.3 Modeling multithreading

The results shown in table 5.4 and in figure 5.3, describe the first approach to understanding multithreading in SMT. In this section, I will not discuss the results obtained by particularly smart tricks as the cyclic reduction for kernel no. 11. As a matter of fact, this discussion is valid only for the loops of independent-iterations and accumulation groups: the model cannot predict any performance improvement to loop-carried-dependence loops.

Two main effects contribute to improve overall performance of multithreaded version of these kernels:

- SMT virtually allows processor's reording capabilities to manage up to 8 copies of the loop-body;

kernel no.	type	size	CPL	loopCPL	actual performance	best proc. util. loop	worst proc. util. ind.	best proc. util. ind.	expected MIN	expected MAX
1	IND	42	18	12	65.23	43.75	42.42	46.67	42.42	43.75
3	IND	24	18	15	16.78	20.00	33.33	66.67	20.00	20.00
4	IND	50	24	24	33.70	26.04	24.51	52.08	24.51	26.04
5	LOOP	31		34	10.40	11.40			11.40	11.40
6	IND	28	50	20	11.20	17.50	11.20	28.00	11.20	17.50
7	IND	93	50	32	50.00	36.33	12.40	22.55	12.40	22.55
8	IND	107	53	48	24.70	27.86	11.88	20.71	11.88	20.71
9	IND	117	190	2	5.00	64.52	3.62	5.03	3.62	5.03
10	IND	130	87	88	31.05	18.47	6.36	16.60	6.36	16.60
11	LOOP	21		18	12.15	14.58			14.58	14.58
12	IND	20	6	2	76.10	93.75	83.33	93.75	83.33	93.75
13	IND	89	38	39	21.76	28.53	13.01	55.11	13.01	28.53
21	IND	34	11	12	59.95	35.42	47.55	85.00	35.42	35.42
23	LOOP	98		111	12.15	11.04			11.04	11.04

Table 5.3: Experimental results, with cc code

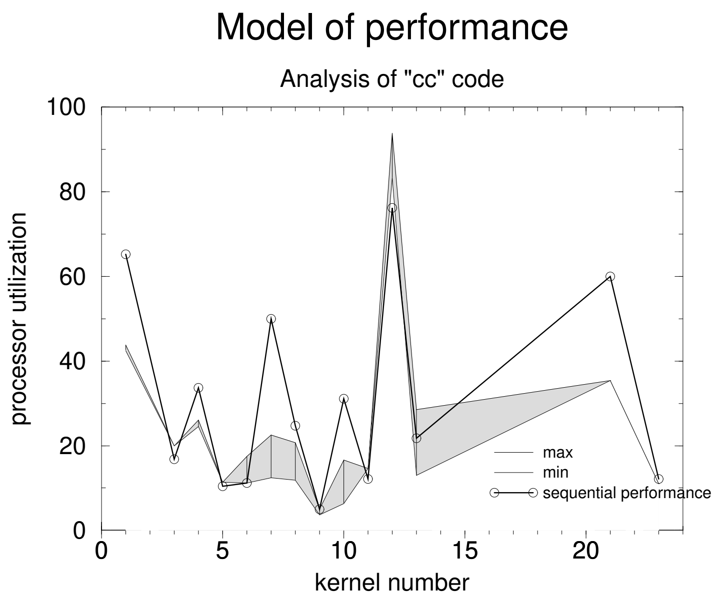


Figure 5.2: Results of modeling performance with cc code

- the need to wait for loop-carried dependences (loop-counters...) is less important as their latency can be hidden too (see increment of `k` in program fragment 3.1).

These two points are important in the following ways: if a kernel is limited by instruction-queue size, with SMT it can count on up to 8 threads from which it can fetch independent instructions. All the values discussed in section 5.2 needs to be recomputed with a number of copies equal to 8 for every kernel.

Please note that, with this model, increasing the number of copies is useful as long as the $saturation\ factor$ is smaller than 1. Consider how it is computed in (5.16): if $saturation\ factor > 1$, we have:

$$saturation\ factor_{max} = \max_{1..5}\left(\frac{number_i * max.copies}{CPL * bandwidth_i}\right) \tag{5.30}$$

$$= \max_{1..5}\left(\frac{number_i}{bandwidth_i}\right) * \frac{max.copies}{CPL} \tag{5.31}$$

Substituting in (5.24), we have:

$$best\ proc.\ util._{ind.} = \frac{size * max.copies}{CPL * \left(\max\limits_{1..5}\left(\frac{number_i}{bandwidth_i}\right) * \frac{max.copies}{CPL}\right) * BW} \tag{5.32}$$

$$= \frac{size}{\left(\max\limits_{1..5}\left(\frac{number_i}{bandwidth_i}\right)\right) * BW} \tag{5.33}$$

that results independent from $copies$.

If a kernel is instead limited by a high value of $loopCPL$, MT version will not be affected in the same way, because the loop critical path can be interleaved with other instructions, as more copies are running in the same time.

Let's try to be clearer: a particulary complex loop-carried dependence, as it can be found in accumulation loops (few floating-point operations, and some integer increments), can be interleaved with instructions from other threads, because they are not waiting for it, as it is the case, instead, with sequential code: its value is required only for the progress of just one thread.

To keep this phenomenon in account, the effect of copies is considered for the loop-carried critical path too. This has a very strong effect for kernel no. 3, 4 and 6.

In figure 5.3, the lower shaded area corresponds to the previous results on sequential performance, while the higher area to the expected MT performance. As before, larger loops are more difficult to model, as it can be seen by the really high performance that the model expects for kernel no. 8 and 13. As said above, large loops' behavior is difficult to understand, and a more accurate model for them is needed.

The proposed model offers good results with small kernels, giving a suggestion of what the speedup of MT code could be. The reader must consider, as a matter of fact, that low performance of specific kernels are due to limited capabilities of the compiler (in this case, myself) to extract all the available parallelism (see for instance the discussion about kernel no. 2 in section 4.4): the model describes an upper bound, that can be reached with good code parallelization (and it is reached — within 4% — by seven kernels in this work: kernel no. 5, 7, 9, 11, 12, 21, 23).

The limiting effects due to resource conflicts and limited memory bandwidth are also grasped: let's consider for instance kernel

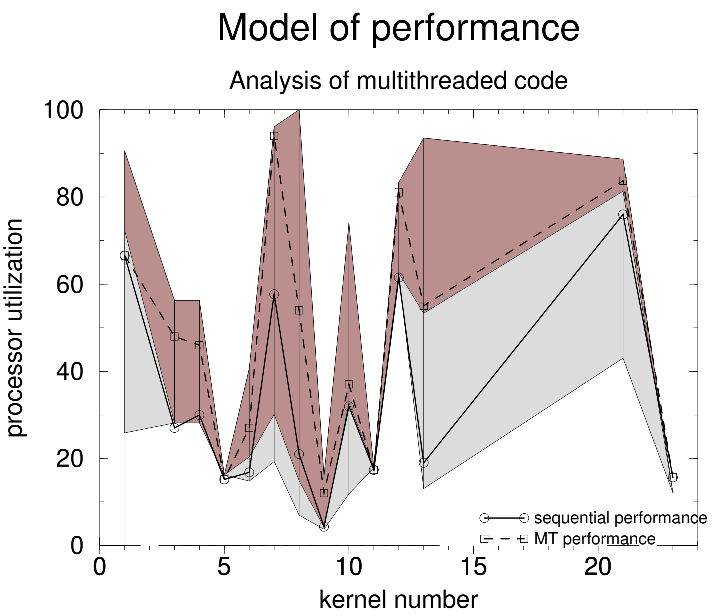


Figure 5.3: Results of modeling multithreaded performance with gcc code

no. 9, sequential performance of which is limited by a very high average memory-access delay. MT versions of it features a better memory utilization and the model can predict this, as it expects a better performance. This is partly explained by the choice of interleaving instead of blocking when rewriting the code, that improves SMT memory utilization in most studied problems (see also (LEL^{+}97)).

5.3.1 Best number of threads

As shown in the previous section, there is a minimum number of copies that allows to reach maximum processor utilization ($saturation\ factor \geq 1$). This can be a way to determine the best number of threads for a specific program, as it is usually useful to keep the number of threads as small as possible in order to have a better utilization of shared resources. With the model, the best number of copies is computed as:

kernel no.	seq. performance	best MT performance	best proc. util. loop	best proc. util. ind.	expected MIN	expected MAX
1	66.50	66.50	100.00	90.48	25.85	90.48
3	27.00	50.00	56.25	96.43	28.12	96.43
4	30.00	52.00	56.25	96.43	28.12	96.43
5	15.20	15.20	16.25	16.25	16.25	16.25
6	16.80	27.00	40.63	40.63	14.77	40.63
7	57.70	94.00	96.09	96.09	19.29	96.09
8	21.00	54.00	100.00	100.00	7.00	100.00
9	4.30	12.00	70.59	16.00	3.76	16.00
10	32.00	37.00	85.00	73.91	11.83	73.91
11	17.40	17.40	17.86	17.86	17.86	17.86
12	61.50	81.00	83.33	83.33	62.50	83.33
13	19.00	55.00	93.48	93.48	13.12	93.48
21	76.00	84.70	88.64	88.64	42.98	88.64
23	15.70	15.70	12.13	12.13	12.13	12.13

Table 5.4: Experimental results, with gcc MT code

$$best\ copies = \begin{cases} \min\limits_{1..5} \left(\frac{bandwidth_i}{number_i} \right) & \text{if indep. iterations} \\ 1 & \text{if dep. iterations} \end{cases} \tag{5.34}$$

This approach has been tried, with the encouraging results shown in table 5.5. Refinement of this model to have a more precise approximation of the best number of threads is surely part of the future work. Anyway, the presented model offers already an estimation of the *trend* that can be observed when more threads are used:

- with kernels featuring loop-carried dependences, adding more threads is not useful when naïve parallelization is used;
- very high expected values describe a situation in which increasing in number of threads is useful; particularly interesting is kernel no. 9, that scales up to 16 threads (see figure 3.2): the model says that it would scale well even to a really higher number of threads.
- low expected values can be observed when the processor-utilization curve features a minimum; these situations require careful tuning of the number of threads.

The most unexpected result is given by kernel no. 6, that seems not to scale very well beyond 4 threads. Its low performance is mostly determined by a very high memory latency, the effects of which lose importance more fast than what described by the model.

Kernel no.	1	3	4	5	6	7	8
Actual	8	2	2	1	4	7	8
Expected	8.8	5.1	5.1	1	17.4	6.3	5.5

Kernel no.	9	10	11	12	13	21	23
Actual	8(16)	7	1	2	5	3	1
Expected	35.3	9.2	1	4	3.3	6.0	1

Table 5.5: Actual best number of threads compared with expected best number of copies

Chapter 6

Conclusion

SMT is a new architecture able to exploit, with the same ease, two different forms of parallelism: instruction-level parallelism, that can be found inside a single thread, and thread-level parallelism, present among different programs running at the same time. This capability is offered by unique SMT's functional-unit sharing among different contexts: SMT can execute instructions from different threads, at the same time, on different functional units.

This thesis has discussed in detail SMT's opportunities for improving performance of MT programs. Two main issues have been addressed:

- the development of some general SMT-specific techniques to parallelize scientific kernels;
- the study of a performance model, able to predict performance for three groups of loops: independent-iteration loops, accumulation loops and loops with loop-carried dependences.

6.1 Parallelization of scientific kernels

Chapter 3 has discussed some techniques to improve MT performance of an interesting scientific benchmark, the Livermore loops. The point of view of an advanced compiler was used, in order to study what the compiling opportunities on SMT are. Main results are:

- *iteration-interleaving* proved to be very good with loop with independent iterations; high average processor utilization (about 65%) is reached; good speedup with up to 8 threads (the maximum in my simulations) was recorded;

- some advanced techniques as *cyclic reduction* and *loop-fusion* proved to be effective, notwithstanding the high overhead that can limit performance when few threads are used; the interesting problem of how to determine the best implementation and the best number of threads have been highlighted;

- accumulation loops resulted particularly interesting as they feature an independent computation, followed by a sequential part; the first part could be easily parallelized, while the second part needed to be protected by some ordering locking; this made versions with few threads more efficient; again, a careful choice of the best number of threads is important; *local accumulation* was also introduced to improve performance in some cases (distributive and associative accumulation);

- larger loops came out not to be parallelizable in many cases, due to the limited knowledge that compilers can extract; naïve techniques (interleaving + locking) are not always effective; in these cases, programmer's help is needed to guide compiling process;

- overall speedup is encouraging: the IPC rate increased by more than 1, just using general techniques; in some cases, up to 3x speedup was observed.

6.2 Performance model

In chapter 5, a performance model for SMT is presented. Using some compile-time parameters, it is able to predict sequential performance, MT speedup and the best number of threads for three groups of loops: independent-iteration loops, accumulation loops and loops with loop-carried dependences.

Main results are:

- the model can predict sequential performance for simple loops (one main basic block): prediction is quite precise (the maximum expected values is no farther than 5% from actual performance) for small loops (less than 40 instructions);

- performance prediction is really good with simple (one main basic block) loop-carried-dependence loops: on average, expected performance is not farther than 1% from actual value (3% in the worst case);
- upper and lower bounds for MT performance is computed by the model; actual performance is always bound by them, and the upper bound is reached — within 4% — by seven kernels in this work;
- a first attempt to estimate the best number of threads is described in section 5.3.1; results offered by the described model are not very precise, but offer already an estimation of the *trend* that can be observed when more threads are used.

6.3 Future works

The parallelizing techniques here discussed need an algorithmical implementation, in order to be introduced into an advanced compiler. First step in this direction will be the development of a set of scripts able to rewrite high-level code.

Also, the presented model, very good with short simple loops, need to be extended to more general cases. A *phased-behavior model* (Smi00) seems to be an interesting opportunity in this direction. Determination of the best number of threads, developed in its first form in section 5.3.1, is a very promising idea that will be discussed more in detail in my future work.

The algorithm used to extract the critical-path length, *CPL*, is at the moment very simple and limited. The introduction of data-flow analysis capabilities will surely improve dramatically its performance. The usage of *MURZ* (an extensible tool for data-flow analysis developed by *Amici Birilli* (Bir99)) to perform this computation is part of the future work.

Furthermore, the present knowledge of engineering trade-offs that SMT introduces is not yet satisfying, in my opinion. As shown in section 2.4, a stronger understanding of chip area usage and printing cost is needed to make SMT a good candidate to be the next processor architecture.

At last, I introduced a broader and more general problem in section 3.7.3: my opinion is that present day ISAs do not exploit some of the hardware characteristics as it could be done. In particularly, I believe that the concept of *registers* should be dropped, because of the decreasing differences in term of performance and functionalities that can be found between architectural registers and data caches. I hope to have the opportunity to spend some time investigating this problem in the near future.

Appendix A

Frequently used acronyms

Here you can find a list of the most frequently used acronyms in this work. Even if any linguist could deplore such a heavy use of acronyms, many of them are very common in Computer Science works, so, here we go.

BTB Branch Target Buffer.

CMP Multiprocessor on a Chip.

CPI Cycles Per Iteration.

CPL Critical Path Length.

CPS Critical Path Schedule.

Dcache Data Cache.

FGMT Fine-grained Multithreading.

FP Floating-Point.

Flop Floating-Point Operation.

Flops Floating-Point Operations Per Second.

Icache Instruction Cache.

IPC Instructions per Cycle.

ILP Instruction-Level Parallelism.

IQ Instruction Queue.

IRAM Intelligent RAM.

ISA Instruction Set Architecture.

MFlops Mega Flops, million of Flops.

MT Multithreaded, Multithreading.

MTA Multithreaded Architecture (©Cray Inc.).

MURZ MURZ (©Amici Birilli 1999).

O^3, OOO Out of order.

PIM Processing In Memory.

PE Processing Element.

RAW Raw Architecture Workstation.

ROB Reorder(ing) Buffer.

SMT Simultaneous Multithreading.

SS Superscalar.

TLB Translation Lookaside Buffer.

TLP Thread-Level Parallelism.

Bibliography

(ABC+97) Gail Alverson, Preston Briggs, Susan Coatney, Simon Kahan, and Richard Korry. "Tera Hardware-Software Cooperation". Supercomputing 1997, page 16, San Jose, CA, November 1997.

(ACCK90) Robert Alverson, David Callahan, Daniel Cummings, and Brian Koblenz. "The Tera Computer System". 1990 International Conference on Supercomputing, September 1990.

(Ald00) Marco Aldinucci. *Smart Memory Parallel Architectures*. PhD thesis, Department of Computer Science, Università degli Studi di Pisa (Italy), 2000.

(Ama00) Saman Amarasinghe. Personal communication, Massachussets Institute of Technology — MIT (Cambridge, MA), March 2000.

(BG00) James Burns and Jean-Luc Gaudiot. "Quantifying the SMT Layout Overhead — Does SMT Pull Its Weight?". 6th International Symposium on High-Performance Computer Architecture, pages 109–20, Toulose, France, January 2000.

(Bir99) Amici Birilli. "MURZ: tutorial and overview". Technical report, Department of Computer Science, Università degli Studi di Pisa (Italy), 1999. Available care of this thesis's author or at: `http://diego.settemari.com/MURZ`.

(BK97) Jay B. Brockman and Peter M. Kogge. "The Case for Processing-In-Memory". Technical Report 10, HTMT, 1997.

(Bre95) Eric Brewer. "High-level optimization via automated statistical modeling". Principles and Practice of Parallel Programming, 1995.

(Com) Compaq. "Alpha C compiler libraries: Atom". On-line documentation.

(CRT99) Brad Calder, Glenn Reinman, and Dean M. Tullsen. "Selective Value Prediction". 26th International Symposium on Computer Architecture, May 1999.

(CT99) Jamison D. Collins and Dean M. Tullsen. "Hardware Identification of Cache Conflict Misses". 32nd Annual International Symposium on Microarchitecture, November 1999.

(DeH96) André DeHon. "Role of Reconfigurable Computing". Panel presentation for reconfig.com RC roundtable, October 1996. Available at: `http://www.cs.berkeley.edu/~amd/reconfig_com_roundtable_oct96/`.

(DeH00a) André DeHon. Personal communication, California Institute of Technology (Pasadena, CA), March 2000.

(DeH00b) André DeHon. "The Density Advantage of Configurable Computing". *IEEE Computer*, 33(4):41–49, April 2000. Special issue on Configurable Computing: Technology and Applications edited by Ranga R. Vemuri and Randoplh E. Harr.

(Die99) Keith Diefendorff. "Compaq Chooses SMT for Alpha". *Microprocessor Report*, 13(16):1, 6–11, December 1999.

(HK97) Mark Howard and Andrew Kopser. "Design of the Tera MTA Integrated Circuits". IEEE Gallium Arsenide Integrated Circuit Symposium, pages 14–17, Anaheim, CA, October 1997.

(Hwa93) Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, New York, NY, 1993.

(Kub00) John Kubiatowicz. Personal communication, University of California (Berkeley, CA), March 2000.

(LEE+97) Jack L. Lo, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, and Dean M. Tullsen. "Converting Thread-Level Parallelism Into Instruction-Level Parallelism via Simultaneous Multithreading". *ACM Transactions on Computer Systems*, pages 322–354, August 1997.

(LEL+97) Jack L. Lo, Susan J. Eggers, Henry M. Levy, Sujay S. Parekh, and Dean M. Tullsen. "Tuning Compiler Optimizations for Simultaneous Multithreading". 30th Annual International Symposium on Microarchitecture (Micro-30), December 1997.

(LPE+) Jack L. Lo, Sujay S. Parekh, Susan J. Eggers, Henry M. Levy, and Dean M. Tullsen. "Software-Directed Register Deallocation for Simultaneous Multithreaded Processors". *IEEE Transactions on Parallel and Distributed Systems*, to appear.

(MCFT99) Nick Mitchell, Larry Carter, Jeanne Ferrante, and Dean Tullsen. "ILP versus TLP on SMT". Supercomputing '99, November 1999.

(McM86) F. H. McMahon. "The Livermore Fortran Kernels: A Computer Test Of The Numerical Performance Range". Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA, December 1986. Available at: National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Road, Springfield, VA, 22161.

(MLM+97) A. J. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings, and Tak Kwan Lee. "The Design of an Asynchronous MIPS R3000 Microprocessor". 17th Conference on Advanced Research in VLSI, Ann Arbor, MI, September 1997.

(MPR99a) *Microprocessor Report*, page 1, October 1999.

(MPR99b) *Microprocessor Report*, page 1, November 1999.

(MPR99c) *Microprocessor Report*, page 11, October 1999.

(MPR99d) *Microprocessor Report*, page 18, October 1999.

(PAC+97) David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. "A Case for Intelligent RAM". *IEEE Micro*, 17(2):34–44, March–April 1997.

(PS96) S. E. Perl and R. L. Sites. "Studies of Windows NT Performance Using Dynamic Execution Traces". Proceedings 2nd Symposium on Operating Design and Implementation, pages 169–183, USENIX Association, Berkeley, CA, 1996.

(RCT+99) Glenn Reinman, Brad Calder, Dean Tullsen, Gary Tyson, and Todd Austin. "Classifying Load and Store Instructions for Memory Renaming". ACM International Conference on Supercomputing, June 1999.

(SFK97) Desző Sima, Terence Fountain, and Péter Kacsuk. *Advanced Computer Architectures — A Design Space Approach*. Addison-Wesley, Harlow, England, 1997.

(SMC+99) Allan Snavely, Nick Mitchell, Larry Carter, Jeanne Ferrante, and Dean Tullsen. "Explorations in Symbiosis on Two Multithreaded Architectures". Workshop on Multithreaded Execution, Architecture, and Compilation, January 1999.

(Smi00) Micheal D. Smith. "Overcoming the Challenges to Feedback-Directed Optimization". ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo 00), Boston, MA, January 2000.

(TEE+96) Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor". 23rd Annual International Symposium on Computer Architecture, Philadelphia, PA, May 1996. Reprinted in Readings in Computer Architecture.

(TEL95) Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. "Simultaneous Multithreading: Maximizing On-Chip Parallelism". 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, June 1995. Reprinted in 25 Years of the International Symposia on Computer Architecture: Selected Papers, 1998.

(TLEL99) Dean M. Tullsen, Jack L. Lo, Susan J. Eggers, and Henry M. Levy. "Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor". 5th International

Symposium on High Performance Computer Architecture, January 1999.

(TS99) Dean M. Tullsen and John S. Seng. "Storageless Value Prediction Using Prior Register Values". 26th International Symposium on Computer Architecture, May 1999.

(Tul96a) Dean M. Tullsen. "Simulation and Modeling of a Simultaneous Multithreading Processor". 22nd Annual Computer Measurement Group Conference, December 1996.

(Tul96b) Dean M. Tullsen. *Simultaneous Multithreading*. PhD thesis, University of Washington, August 1996.

(WCT98) Steven Wallace, Brad Calder, and Dean M. Tullsen. "Threaded Multiple Path Execution". 25th Annual International Symposium on Computer Architecture, June 1998.

(WTC99) Steven Wallace, Dean M. Tullsen, and Brad Calder. "Instruction Recycling on a Multiple-Path Processor". 5th International Symposium on High Performance Computer Architecture, January 1999.

(WTS+97) Elliot Waingold, Micheal Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. "Baring It All to Software: Raw Machines". *Computer*, pages 86–93, September 1997.

(Yea96) K. C. Yeager. "The MIPS R10000 superscalar microprocessor". *IEEE Micro*, pages 28–40, April 1996.

...ho sentito che ti stai per laureare in Informatica... ascolta, posso chiederti una cosa? Quando navigo in Internet, ed apro un sito, mi dice che c'è un problema con i cookies... cosa devo fare?

Anonymous

Pisa, June 30, 2000