Inthreads — a new computational model to enhance ILP of sequential programs

Alex Gontmakher, Assaf Schuster, Gregory Shklover
(g Sasha, assaf, gkovriga)@cs.technion.ac.il

Avi Mendelson
avi.mendelson@intel.com

Abstract

This paper presents Inthreads, a new computational model based on sharing the register file that enjoys the benefits of multithreading within the context of a single SMT thread. The model incurs overhead that is orders of magnitude lower than that of conventional threads and therefore supports fine grain parallelization.

We present the basic concepts of the Inthreads model, the compilation process, and describe the mechanisms that provide an efficient implementation of thread management and synchronization operations. We take special care of instruction dependencies that cross thread boundaries due to the register sharing.

Our simulation results indicate that fine granularity threads can be used for different single threaded kernels and applications. We reach speedup of up to 2.5x on benchmarks from SPEC and up to 4x on individual code sections. In cases where the memory is the performance bottleneck, we have been able to hide an increase of memory latency from 100 to 400 cycles.

1 Introduction

Recent developments in semiconductor technology indicate that, unless fundamental breakthroughs are made, we can no longer expect the steady exponential increase in clock frequency. Thus, if we like to preserve the rate of performance improvement that has been sustained for several decades, we will have to increase the average number of instructions the processor can execute in parallel.

The approaches proposed for improving the parallelism can be classified into two major groups: explicit, which involve parallel code written by the user and implicit, where the architecture unravels parallelism out of a sequential program by using hardware techniques such as out-of-order execution.

Out-of-order computers expose the parallelism in single-threaded code essentially by sliding a window of instructions over the computation sequence. The order in which instructions in the window can execute is determined by their data dependencies. The maximum parallelism achievable by an out-of-order processor depends on the amount of dependencies between the instructions, the size of the instruction window and other hardware limitations such as branch prediction quality and number of execution units. However, research indicates that for a reasonable window size, the amount of parallelism within the window that contains instructions from a single thread grows as \( \log(\text{window size}) \) [12]. Therefore, increasing the window as a means of improving IPC is practical only to a certain degree.

Multithreading can be seen as a way for genuinely increasing the number of independent instructions within the window. We can view the instruction windows of a multi-threaded architecture as one aggregated window working on several places of the code in parallel. The amount of independent instructions for a given window size increases almost linearly with the number of threads, provided that most of the time, instructions fetched from different threads are independent.

Many current architectures [11, 8] include support for Simultaneous Multithreading [16] to allow efficient execution of multithreaded code, introducing such capabilities as parallel fetching for several threads, private Reorder Buffers etc. In this paper, we would like to take advantage of these mechanisms for parallelization of sequential code. To this end, we present a new computational model in which portions of sequential code is broken into micro-threads, termed inthreads, that share the execution context of a single SMT thread. Sharing of the execution context allows inthreads to communicate through registers, which is inherently faster.
than communication through memory. Moreover, the architecture allows for all the thread management to be integrated in hardware, minimizing the overhead of thread creation and synchronization.

The Inthreads model is able to perform thread creation, termination, synchronization and communication at a rate comparable with execution time for instructions in a instruction window, and thus enjoys the benefits of multithreaded execution at the granularity of superscalar execution. This opens a new venue for expressing the compile-time knowledge of available parallelism and thus presents a new dimension of optimizations to the compiler.

This paper introduces the fundamentals of the Inthreads architecture and describes the basic implementation mechanisms. The architecture presents a row of new issues and opportunities, such as code generation, speculative execution of synchronization instructions and more. We intend to cover these issues in the future work, based on the results presented here. Our major contributions in this paper are the following:

1. We describe an architecture that is capable of efficiently performing communications between threads at extremely low granularity. Due to management in hardware, our architecture is several orders of magnitude more efficient than conventional multithreading.

2. We show that there is significant potential for parallelization by low-granularity threads in some problematic codes.

In this paper we use the term thread to denote a thread in the Inthreads model, and SMT thread to denote a conventional thread.

The rest of the paper is organized as follows. In Section 2 we describe the architectural support for our programming model and explain how it provides better performance. In Section 3 we describe the microarchitectural mechanisms necessary to obtain high performance from inthreads-targeted programs. Section 4 presents the simulation results. Finally, Section 5 discusses the related work and Section 6 summarizes the work and concludes.

2 Programming Model

2.1 ISA Extension

The Inthreads computational model is based on a fixed number of threads running over shared architectural registers in a context of a single SMT thread. As a result, the model provides an extremely lightweight threading mechanism: the fixed number of threads allows for thread management to be performed completely in hardware. In addition, the shared registers provide a straightforward and efficient communication mechanism: a value can be transferred between threads by writing it into a register in one thread and reading it from the same register in another.

The sharing involves all the general-purpose registers except for those necessary to maintain the private state of each thread. The most important register in the private state is the Program Count (PC) register. Several additional registers need to be thread-private; due to space limitation, we omitted the description and the implementation of these registers.

Each thread has a numeric thread ID (TID) which is determined at thread creation. The main thread, identified by TID=0, is always active, while other threads can be started and terminated on demand.

To start and stop threads, we provide three new instructions: \texttt{inith.start}, \texttt{inith.halt} and \texttt{inith.kill}. \texttt{inith.start} creates a new thread with a given thread ID at a given address, specified by the ID and OFFSET parameters. To terminate itself, a thread issues an \texttt{inith.halt} instruction. \texttt{inith.halt} is executed synchronously, guaranteeing that all the preceding instructions complete. Killing another thread is requested by a \texttt{inith.kill} instruction, which receives the ID of the thread to be killed.

The synchronization mechanism consists of a set of binary semaphores stored in condition registers and three new instructions: \texttt{cond.wait}, \texttt{cond.set} and \texttt{cond.clr}. All the synchronization instructions receive a single parameter, the condition register to access.

A \texttt{cond.wait} checks whether a given condition is set. If it is, the condition is cleared, otherwise the issuing thread is stalled until some other thread performs a \texttt{cond.set} to that condition. If several \texttt{cond.wait}
instructions accessing the same condition are issued in parallel, only one of them will proceed. A cond.set sets the given condition. If there was a cond.wait suspended on the same condition, the cond.wait is awakened and the condition remains cleared. Finally, a cond.clr clears the given condition.

A fundamental requirement of processor architecture is that the result of any execution must maintain the illusion of sequential and atomic execution. A conventional processor uses the ROB to hide the out-of-order execution of instructions. An inthreads processor needs additional mechanisms since the register sharing exposes the out-of-order processing. The architecture achieves sequential semantics by collaboration between the hardware and the software. On the hardware side, the microarchitecture ensures correct transfers of values for the accesses to the shared registers. On the software side, the compiler must ensure the absence of conflicting register accesses. To this end, the private variables of different threads must use disjoint subsets of the register file, and the accesses to the shared variables must be free of data races, protected by synchronization if necessary.

The avoidance of data races, though requiring increased compiler complexity, simplifies the hardware considerably. Handling the consequences of conflicting accesses would either involve excessive serialization of execution or require expensive rollback mechanisms to handle violations of sequential execution. The absence of data races makes these measures unnecessary; it is enough to implement Release Consistency \cite{9} with respect to register and memory accesses to ensure sequential execution semantics.

The Release Consistency distinguishes strong operations (Release and Acquire, which maintain the order with respect to all prior and following operations, respectively) from regular weak operations. In our model, the Release semantics is required for cond.set and int.start instructions and the Acquire semantics is required for the cond.wait instruction. The implementation of the semantics is discussed in Section 3.

Although register sharing is a cornerstone of our new computational model, it has some potential drawbacks. One drawback is the interference between parallelization and the function calling mechanism. Most calling conventions involve saving the registers on the stack and restoring them upon returning. This technique is based on the implicit assumption that the calling frame is inactive while the callee is executed. However, this assumption is incorrect in the Inthreads model: if a thread performs a call while other threads are active, a register access conflict may arise.

In many cases this problem can be mitigated by function inlining which exposes enough parallelism for inthreads parallelization to be applicable. In addition, we envision several possible methods which would allow function calls in parallel code. One method involves virtualizing the registers, similarly to the Register Stack Engine of IA-64, so that each function call actually works on different registers. Another one is based on suspending execution of threads in order to perform a function call. Results presented here are based on inlining only and therefore these solutions are out of scope of this paper.

Another consequence is the result of fewer registers being available to a thread. In general, threads in the inthreads model have slightly lower register demand than SMT threads due to the sharing of common registers, such as SP, GP and registers holding read-only values. Still, parallelizing to a high degree (above four threads) may increase the register pressure to the point that extensive spilling is required, degrading the performance. This phenomenon effectively limits the practical number of threads to a constant. This limit is not inherently constraining since the requirement of hardware management for threads implies fixed number of threads anyway. Still, we found that increasing the number of architectural registers beyond 32 would benefit the performance of an Inthreads processor.

### 2.2 The Process of Code Generation

Ideally, a compiler would transform the source code directly into Inthreads-enabled assembly. However, building the compiler in one step is undesirable. The process of code generation is quite complex and presents challenges at several levels. First, we must identify opportunities for parallelization, i.e., determine the sections of the code with poor instruction-level parallelism that can be partitioned into threads. Second, we need to apply the parallelization to the identified cases. Third, we need to generate the machine code according to the Inthreads programming model.

To facilitate experimentation and separate the concerns of parallelization from the technical details of low-level code generation, we splitted the process into two stages, the parallelization and the actual code
generation, illustrated in Figure 1. As an interface between the stages, we defined an intermediate language, Inthreads-C, which extends the C programming language with Inthreads-specific constructs.

The input for Inthreads-C can be generated either by a parallelizing compiler or manually. The results presented in this paper are based on manual parallelization, however, we believe that the code transformations we employed are applicable for automatic parallelization as well. We intend to present our work on automatic parallelization in one of our future papers.

The semantics of Inthreads-C require the compiler to be aware of all the code that can be running simultaneously in different threads. To this end, we encapsulate the code of each intthread in a block marked by a special #pragma intthread statement. Using this information, the compiler can find all the blocks that are reachable by intth.start commands from a given point to determine the threads that are running in parallel.

The presence of register-shared code in Inthreads-C has a profound effect on the code generation. For instance, if a variable is written in one thread and read in another, the compiler must recognize that the writing instruction is not dead code while there is no obvious control flow path from the writing instruction to the reading one.

We have implemented the Inthreads-C compiler by extending GCC 3.4 with the Inthreads features. The extensions affect both the compiler frontend, which was modified to recognize the Inthreads constructs, as well as some of the optimization passes, which must be thread-aware in order to function correctly. A complete description of the compiler will be provided in a forthcoming paper.

Figure 2 presents an example of Inthreads-C code, which is a simplified version of code in the 181.mcf benchmark from SPEC2000. Parallelization of this code with seven threads achieved 4x speedup in our experiments. The complete results are presented in Section 4.4.

The original source code is shown in Figure 2a. The invocations of (1), (2) and (3) in different iterations are independent and can be executed by multiple threads. Different invocations of (5), (6) can also be executed in parallel to each other and to those of (1), (2) and (3) as long as the correct value of next is known. The only part of the code that must be serialized is the updating of next in (4).

The parallelization proceeds as follows. We create a coordinating thread, shown in Figure 2c, and several worker threads which carry out the iterations in a round-robin manner. The first worker is shown in Figure 2b.

Since each worker needs to use the variables a and r independently, we need private copies of those variables, a_p and r_p. In addition, variables res1 and d1 are used for communication between threads. Since res1 and d1 are shared, we protect the accesses to them by synchronization, using condition registers 1 and 2. A sequence of cond.set 1-cond.wait 1 is used to send the value of res1 to the coordinating thread, and a sequence of cond.set 2-cond.wait 2 to get back the value of d1 Finally, we reuse condition register 1 to notify the coordinating thread of the completion of the worker.

3 Implementation

In principle, the architecture of Inthreads is quite similar to SMT [16]—both architectures are based on multiple independent streams of instructions which are mixed in a shared execution core. Therefore, the microarchitecture re-uses most of the mechanisms present in SMT processors, such as multiple fetch units, multiple Reorder Buffers (ROBs), shared physical register file and functional units and so on. A notable exception is the Register Allocation Table (RAT), which is shared between threads in an Inthreads processor. In addition, the Inthreads architecture requires mechanisms to support efficient execution of synchronization operations. Figure 3 presents these mechanisms (the Inthreads-related ones are shaded).

The Wait Buffers (WB) implement the delaying of cond.wait instructions for which the required condition
Figure 2: Example of parallelization in Inthreads-C.

Figure 3: Inthreads implementation outline.
is not ready. The delaying is performed independently for each thread, and therefore there is one WB per intthread. To accommodate the overhead, a new pipeline stage is dedicated for the WBs. Note that the Wait Buffers are placed before the renaming logic of the processor. Therefore, the register dependencies, which are defined during renaming, follow the semantics of synchronization.

The Thread Control Unit (TCU) orchestrates the execution of the threads on the processor. The TCU receives all the thread-related instructions, and issues the necessary control signals to apply their side effects. First, the TCU notifies the Wait Buffers when conditions become available, allowing the corresponding cond.wait instructions to proceed. Second, it instructs the Fetch unit to start and stop fetching of instructions in response to int.start, int.kill and int.halt instructions.

The Condition Speculation Table (CST) keeps track of the unresolved branch instructions. This information is used by the TCU to determine the speculative status of instructions.

### 3.1 Execution of thread control and synchronization instructions

The execution of the thread-management instructions proceeds in the following way. All the thread-related instructions are diverted to the TCU for execution. To carry out the thread management instructions (int.start, int.kill and int.halt), the TCU must notify the Fetch stage to start or stop fetching instructions for the corresponding thread. For an int.start instruction, it sends the TID and the address of the new thread to the Fetch stage using the Started Threads line. For an int.kill or an int.halt, it sends the TID of the killed thread using the Killed Threads line.

The scheduling of int.halt and int.kill is different. A int.kill, which is executed asynchronously with respect to the instructions of its own thread, is released immediately to the TCU. Contrastingly, a int.halt is executed synchronously, i.e., it must ensure that all the preceding instructions complete and none of the following instructions get executed. To this end, a int.halt is locked in the WB until all the preceding instructions complete.

The synchronization instructions operate in the following way. When a cond.wait instruction is decoded, it is sent to the corresponding WB. The Wait Buffer monitors the Available Conditions line to determine when the cond.wait contained in it can be released. To keep sequential execution within each thread, when a WB is not empty, all the freshly decoded instructions of that thread are diverted to the WB as well. To avoid overflowing, a non-empty WB requests the Fetch Unit to delay fetching instructions of the corresponding thread by signalling on the Delayed Threads line.

The values of the condition variables are computed by the TCU, which updates them according to the received synchronization instructions.

### 3.2 Handling Speculation

The register sharing exposes data dependencies between instructions issued by different threads to speculation. If an instruction is squashed and a dependent instruction in another thread is not, the dependent one will receive a wrong value or never receive a value at all. Our handling of such situation involves collaboration of the software and the hardware.

On the software side, the data-race-free requirement ensures that any pair of dependent instructions by different threads is separated by a synchronization sequence. On the hardware site, the implementation must ensure correct interaction of speculation with synchronization. One approach is to avoid issuing the cond.set instructions speculatively. This prevents instructions from receiving speculative values from other threads, but incurs some overhead due to delaying of execution. Another approach is to allow speculative execution of cond.sets and introduce a mechanism to identify and squash the dependent instructions from other threads. The results presented in this paper are based on the first approach and are therefore somewhat pessimistic.

The computation of speculative status is based on instruction timestamps that provide per-thread enumeration of instructions currently residing in the ROB. The Control Speculation Table (CST), shown in Figure 4 stores the timestamps of all the unresolved branches. The first timestamp in the CST for a given thread identifies the first unresolved branch of that thread. An instruction is speculative if its timestamp is later than the timestamp for its thread provided by the CST.
The CST is updated during each cycle. The timestamps for the freshly decoded branches are sent to the CST and stored there. In addition, the CST receives information on execution of branches: all the resolved branches and those found on a mispredicted path are discarded.

The Thread Control Unit (TCU), shown in Figure 5, keeps two separate queues for the thread management and the synchronization instructions (TMQ and CIQ, respectively). Incoming instructions are dispatched into the appropriate queue. Instructions in both queues are issued as soon as they become non-speculative. To determine this, the TCU keeps timestamps for all its instructions and compares them to the timestamps received from the CST. Similarly to the CST, the TCU receives the information on all the resolved branches and discards all the instructions found on mispredicted paths.

When issued, the thread management instructions merely assert the necessary signals, as described in Section 3.1. The synchronization instructions update the status of the currently available conditions.

The conditions are computed from the results of all the committed synchronization instructions (stored in the Committed Conditions Register (CCR)) and from the instructions currently waiting in the CIQ. Only some of the instructions in the CIQ affect the computed conditions. A cond.wait or a cond.clr must reset the resulting condition to make sure that only one cond.wait wakes up for each cond.set. In contrast, a cond.set instruction the CIQ is speculative and therefore should not affect the conditions. The value of the currently available conditions is therefore computed by cleaning from the CCR all the conditions corresponding to the cond.waits and cond.clrs in the CIQ. The values in the CCR are updated by all the issued instructions.

Note that when a cond.wait or a cond.clr is squashed, the condition held by it is automatically restored when the available conditions are recomputed.

The Condition Queue size limits the number of synchronization instructions that can be in flight at the same time. If the queue overfills, the decoding must be stalled. However, we have found that with a queue size of 16 the number of such stalls is negligible.

## 4 Evaluation

We have extended the SimpleScalar-PISA model [5] to include the implementation of the new instructions and the microarchitectural extensions. The basic processor is modelled with a 8-stage pipeline. The Inthread-enabled version adds one stage after Decode to accommodate the synchronization functionality. Except where stated explicitly, the processor front-end was configured to fetch up to 8 instructions from up to 4 concurrent threads, managed by the ICOUNT policy. Table 1 lists the relevant processor parameters.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>No threads</th>
<th>Intreads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline Length</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>Supported threads</td>
<td>N/A</td>
<td>8</td>
</tr>
<tr>
<td>Fetch policy</td>
<td>8 per cycle</td>
<td>TCOUNT4.8</td>
</tr>
<tr>
<td>Branch predictor</td>
<td>bimodal</td>
<td></td>
</tr>
<tr>
<td>L1I size</td>
<td>64KB</td>
<td></td>
</tr>
<tr>
<td>Logical Registers</td>
<td>64GP+64FP</td>
<td></td>
</tr>
<tr>
<td>Physical Registers</td>
<td>512GP,512FP</td>
<td>384GP,384FP</td>
</tr>
<tr>
<td>ROB size</td>
<td>512</td>
<td>128*8</td>
</tr>
<tr>
<td>Issue Queue size</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>Memory Queue size</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>Functional units</td>
<td>6 Int,6FP,4 Branch</td>
<td></td>
</tr>
<tr>
<td>Max. outstanding misses</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Memory ports</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>L1D size</td>
<td>64KB</td>
<td></td>
</tr>
<tr>
<td>L2 size</td>
<td>1MB</td>
<td></td>
</tr>
<tr>
<td>L2 latency</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Memory latency</td>
<td>200</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Basic processor parameters.

The evaluation is based on three benchmarks from the SPEC2K suite [10]: Art, Mcf and Twolf and several computational kernels designed to demonstrate the potential and the limitations of our model. Some of the other spec benchmarks, such as Crafty, are inherently hard for Intreads parallelization because they spend most of the time in recursive calls. Others, such as bzip, have code that is mostly parallelizable but includes intermittent system calls, and thus need hardware support for calls in parallelized code, not available at this time. The maximal speedups achieved by the benchmarks are shown in Table 6. ASSAF,AVI: Please look, this is probably the most important place in the paper!!

In the benchmarks, we have applied the parallelization only to those portions of the code that exhibited poor performance due to high cache or branch predictor miss rate (and constituted a significant part of the program's run time). The programs were run until completion, with reduced inputs that required about 1 billion instructions. To measure the program running time we used the number of clock cycles reported by the simulator.

### 4.1 Intread Management Overhead

In order to compare the overhead of the Intreads parallelization to that of conventional parallelization, we parallelized a small function with two tight loops that can run in parallel. The Intreads parallelization was performed by spawning a thread to run one of the loops. The conventional parallelization was performed through pthreads package. Two parallelization approaches were attempted. In the first approach, a thread was created upon program startup, and received input from the main program through synchronization. In the second one, a new thread was created upon function entry. Both pthreads experiments were executed on a dual PPC computer running Linux 2.6.5.

Figure 7 shows the speedup achieved by parallelization as a function of the number of iterations in the loop (note that the X axis is logarithmic). We can see that at the loop size of 100 iterations the Intreads parallelization reaches its complete speedup potential, while the pthreads variant exhibits a slowdown of over 10x, reaching the break-even point only at about 3000 iterations. The Intreads overhead is about three orders of magnitude lower than that of pthreads.

Note that this experiment measures only the overhead of thread manipulation. If we would use addi-
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Parallelization Factor</th>
<th>Speedup</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>159.art</td>
<td>7</td>
<td>1.50</td>
<td>Discussed in Section 4.3</td>
</tr>
<tr>
<td>181.mcf</td>
<td>7</td>
<td>2.36</td>
<td>Discussed in Section 4.4</td>
</tr>
<tr>
<td>300.twolf</td>
<td>2</td>
<td>1.09</td>
<td>27% better IPC, 15% more instructions</td>
</tr>
<tr>
<td>186.crafty</td>
<td>N/A</td>
<td>N/A</td>
<td>Hard to parallelize due to recursive calls</td>
</tr>
<tr>
<td>164.gzip, 176.gc,c, 256.bzip2</td>
<td>N/A</td>
<td>N/A</td>
<td>Need support for calls in parallelized code</td>
</tr>
<tr>
<td>chase</td>
<td>8</td>
<td>3.88</td>
<td>Kernel: Pointer chasing</td>
</tr>
<tr>
<td>count</td>
<td>8</td>
<td>2.95</td>
<td>Kernel: Heavy branching with poor prediction</td>
</tr>
<tr>
<td>wife</td>
<td>8</td>
<td>1.92</td>
<td>Kernel: Saturates the processor, no opportunity for speedup</td>
</tr>
</tbody>
</table>

Figure 6: Summary of the evaluated benchmarks.

Figure 7: Speedup achieved by the minimal parallelized code.
tional Inthreads mechanisms, such as communication through shared registers, the gap to the conventional parallelization could be even more pronounced.

4.2 Computational Kernels

To demonstrate the potential benefits and limitations of inthreads parallelization, we have used three microbenchmarks: count, chase and wide with different patterns of instruction dependencies, memory accesses and branching behavior.

The Chase benchmark performs pointer chases on multiple linked lists, and has a high L1 and L2 miss rate. The speed of the benchmark is limited because each load needs the result of the previous one to get its address. The parallelization achieves speedup by carrying out several pointer chases in parallel, resulting in concurrently outstanding loads and thus improving the utilization of the memory subsystem. As can be seen in Figure 8, the benchmark enjoys almost linear speedup until it saturates the memory. Note that the proportion of the squashed instructions does not change with parallelization degree. The reason is that the benchmark affects only the parallelism of memory accesses, and the success rate of the speculation remains unchanged.

The Count benchmark counts the cleared bits in random data, featuring a high branch misprediction rate. The branching behavior is the major factor determining the speed of the benchmark. First, the fetch bandwidth is limited by the speed of the branch predictor. Second, the number of instructions that reach the retirement is limited by the high misprediction rate and would only increase slightly if the processor is able to fetch and execute more instructions speculatively.

Parallelization takes care of both effects. The branch prediction, which is independent for different threads, can be executed in parallel and thus provide a higher fetch bandwidth. Furthermore, as the parallelization degree grows, the number of instructions squashed as a result of misprediction is reduced, resulting in an increased fraction of fetched instructions that reach retirement. This effect can be seen in Figure 8b: the parallelization leaves the absolute number of squashed instructions unchanged, while the number of committed instructions grows. The speedup of Count peaks when the code saturates the fetch bandwidth, therefore, there is no additional speedup after 5 threads.

In contrast to Chase and Count, the Wide benchmark has a good memory and branch behavior, and consequently has high utilization of the processor resources. Still, as Figure 8c shows, parallelization helps the benchmark reach higher resource utilization, mostly due to the improvement in fetch bandwidth and to a lesser degree due to reduction in the number of squashed instructions.

4.3 Discussion — 179.art

In 179.art, threads are created to perform relatively small chunks of computation, on the order of several thousand instructions each. Parallelizing at larger granularity is impossible, as each chunk depends serially on the result of the previous one. The overhead of starting and terminating an SMT thread would be comparable to the benefit of parallelization, rendering the conventional multithreading inapplicable in this case. Serialization in 179.art is applied at a relatively coarse degree. It is used mostly to receive the results of the worker threads and wait for their completion.
The performance results for the complete execution of 179.art are presented in Figure 9a (the speedup of individual code sections is significantly higher, up to 2.5x at 6 threads). We can see that the performance improvement is achieved by optimizing the fetch bandwidth without increasing the absolute number of squashed instructions (and actually decreasing their proportion). This implies that the performance is dominated by the behavior of fetching and speculative execution. Indeed, as Figure 9b shows, the speedup of parallelization improves with more aggressive fetch policy.

Figures 9c and d show the speedup as a function of the memory latency. We can see that while the execution time is affected strongly by the latency, the speedup barely changes. This implies that 179.art contains little pointer chasing, and thus the behavior of memory subsystem is not affected by parallelization.

To assess the sensitivity of the parallelization to the register file size, we have compiled the benchmark with 32 and 64 registers. The result, shown in Figure 10a, indicates that the compiler finds enough registers for up to 4 threads, after which the performance deteriorates quickly due to excessive spilling. Parallelization with 6 threads actually causes a slowdown on 32 registers.

4.4 Discussion — 181.mcf

The parallelization in 181.mcf is more involved than that of 179.art. Most of the parallelization is performed in function train.match, a simplified version of which is shown in Figure 2. As explained in Section 2.2, the code contains a small portion of serial code which nevertheless requires intensive synchronization, with frequency of about 0.7 synchronization instructions (and the related data transfers) per cycle. Each synchronization involves also a data transfer between threads; communicating this often would severely impact
the performance. Still, thanks to data transfer through registers, we are able to achieve a speedup of up to 4x in the parallelized code.

The performance results of 181.mcf are presented in Figure 11a. We can see that the percentage of squashed instructions remains almost constant, indicating that the bottlenecks of the execution are not related to the fetching and speculation behavior. Indeed, as Figure 11b shows, changing the fetch policy has little effect on speedup. On the other hand, the speedup does increase with the memory latency, as shown in Figure 11c. This is not surprising considering the amount of double pointer indirection in the code.

Figure 11d shows the execution time as a function of memory latency. It is interesting to note that execution time of the parallelized version on a processor with memory latency of 400 cycles is almost the same as that of the serial code at memory latency of 100 cycles.

Behavior of mcf with respect to the number of registers is better than that of art, as can be seen in Figure 10. When the parallelization reaches 5 threads, the compiler runs out of registers and spills several heavily used variables. With 6 threads, spilling increases only slightly, and the result improves again due to higher parallelization degree.

5 Related Work

This work is based on the observation that programs often do not need many registers in order to reach top performance [4]. A natural way to take advantage of this fact is to divide the registers between several threads, gaining speedup by parallelisation while hurting only slightly the performance of each thread.

Several researches have investigated methods for sharing registers between threads under compiler control. The most closely related work is the mSMT proposal by Redstone et al. [14], which adds multiple thread contexts for each SMT thread. The architectural register file is divided equally between registers; no communication is performed through registers. Redstone et al. examine several inherently parallelisable programs and observe an average 38% improvement in throughput with two contexts, with only 5% average slowdown of each thread. A comparable approach can be found in the work of Waldspurger and Weihl [18]. In contrast, in our work, the threads working in context of a single SMT thread may access all the registers. The threads use the shared registers as an extremely lightweight communication mechanism, which is used for parallelization by threads at the level of sequential code.

Several researchers have investigated parallelizing programs with microthreads in order to tolerate the
memory latency. The Threadlets architecture by Rodrigues et al. [15] splits basic blocks into threads using a special trace-based compiler. The approach, applied to several scientific benchmarks, achieves 80% of the potential parallelism in most cases. In comparison, our model presents a more general threading mechanism, with control flow and generic communication capability, which is applicable to less uniform programs such as SPEC benchmarks.

Another method of tolerating long-latency accesses is to switch threads immediately when such an event occurs. Mowry and Ramkisson [13] propose a system in which the processor traps into a thread-switch routine immediately after discovering a cache miss. To reduce the context switch overhead, the researchers use the compiler to statically partition the registers between the threads.

Fine-granularity threading requires a fast hardware based synchronization mechanism. This mechanism usually takes a form of Full/Empty bit (FEB), which can be associated with memory locations [1, 2] or a register-like lock box [17]. The latter work achieves speedup for parallelising programs which slow down when executed on a conventional architecture. Our work enhances this approach by introducing register sharing, allowing us to perform even finer grain parallelization.

Other works on fine-grain multithreading use threads that assist the main thread by warming up the cache and branch prediction or simply running forward speculatively to explore speculative paths to a considerable depth [7, 3, 6]. In these works, the communication between threads is either completely unnecessary, or is performed only in one direction, from the parent thread to the spawned one.

6 Conclusions

In this work, we have presented intthreads, a new computational model that achieves the benefits of multithreading within the context of a single SMT thread. The model is based on a shared register file, resulting in extremely lightweight thread communication and synchronization that allow the compiler to parallelize code of single-threaded applications. The implementation utilizes the SMT implementation mechanisms already available in many modern processors and adds the logic that implements intthreads-related instructions.

Our results show that with an efficient implementation, threads can communicate and synchronize at an extremely high rate, on the order of one synchronization operation per cycle. This capability allows us to achieve speedups by parallelizing code which is otherwise serial, impossible to parallelize using conventional mechanisms. We have achieved speedups of up to 4x in microbenchmarks and up to 2.5x on benchmarks from SPEC. In one case, the parallelization was able to mask an increase of memory latency from 100 to 400 cycles.

This paper sets the basis for discussion of the Inthreads programming model. The model raises numerous issues with code generation, speculative execution of synchronization instructions, memory reordering based on the knowledge of synchronization dependencies and others. These issues will be covered in our future publications.

References


