

Intrathreads: Techniques for parallelizing Sequential Code

Authors: Alex Gontmakher, Assaf Schuster

Keywords: Parallelization, Multithreaded, Code Optimization, Lightweight Synchronization, Register Sharing

Abstract

The *inthreads* architecture enables low-level parallelization of serial computation. This paper describes the inthreads architecture and shows several code transformations that can be used for optimizing code with low instruction-level parallelism. Such code can be optimized neither with conventional techniques due to complex branching, nor with conventional concurrent programming due to very low granularity of the parallelizable code sequences.

Corresponding Author: Alex Gontmakher

email: gsasha@cs.technion.ac.il

phone: (972) 89404782

surface mail: HaKeshet 3/3, Ness-Ziona, Israel

Intrathreads: Techniques for Parallelizing Sequential Code

Alex Gontmakher, Assaf Schuster

gsasha, assaf@cs.technion.ac.il

Abstract

We present an architecture that enables low-level parallelization of serial computation using a special kind of threads called *inthreads*. Inthreads adapt the techniques and algorithms of concurrent shared-memory programming for the computation stream of regular code. The architecture is simple and light-weight, needing only six instructions to manipulate the inthread mechanisms. The architecture is designed to complement, rather than replace, conventional parallelization techniques, such as OS-based threads, SMT, chip multiprocessing and others.

This paper describes the inthreads architecture and shows several code transformations that can be used for optimizing code with low instruction-level parallelism. Such code can be optimized neither with conventional techniques due to complex branching, nor with conventional concurrent programming due to very low granularity of the parallelizable code sequences.

To evaluate the benefits of the architecture, we have built an execution-driven simulator. The results of simulation indicate that inthreads can provide nearly-linear speedups in the number of inthreads used.

1 Introduction

Modern processors perform many operations in parallel in order to achieve high performance. To realize the benefit of parallelization, high degree of resource utilization must be achieved. However, a major road block to high efficiency is posed by data and control dependencies. Long sequences of branch instructions or memory fetches are quite common in current programs. Such sequences cannot be parallelized at the instruction level, because of their dependencies. On the other hand, the computation can often be divided into small chunks that can be executed independently. Unfortunately, those chunks are usually

```
int size=0;
for (Node* n=first; n!=0; n=n->next)
    if (n->data!=0) size += strlen(n->data);
```

Figure 1: CALCULATING THE SUM OF LENGTHS OF STRINGS IN A LINKED LIST.

too fine-grained to be efficiently utilized using conventional multithreaded programming techniques.

For a motivating example, consider the code in Figure 1, which traverses a linked list and sums up the lengths of the strings in it.

The `for` loop performs two branches per iteration, therefore it will not be able to exhaust the processor resources. Similarly, the computation of `strlen` usually consists of a `while` loop with very little work to be done on each iteration, therefore it too will not be able to fill up the processor pipeline. Running the loop traversal concurrently with the computation of `strlen` could achieve higher utilization of the processor, leading to better performance.

It is very hard to write sequential code that will enable parallelization at this level, because both parts of the computation include complex branch patterns and potentially high memory latencies. On the other hand, spawning a thread for every computation of `strlen`, or just using a second working thread that would run these computations in parallel to the list traversal, will incur overhead of thread creation and synchronization which is by itself heavier than the code in question.

To solve this problem, we need a threading mechanism with extremely low overhead of thread creation, synchronization, communication and switching. From the point of view of the operating system, the mechanism must work transparently within the context of a single OS thread. The processor would provide several streams of control and issue their instructions in an interleaving fashion, eliminating the need for context switch.

We call this mechanism *Intrathreads*, or *inthreads* for short, to stress the fact that it works within the

processor.

The inthreads mechanism is simple and generic. It defines only a few additional instructions and can be implemented with little overhead in processor resources. The focus of this paper is to study the techniques that employ inthreads for optimization of code.

The rest of this paper is organized as follows. Section 2 describes the architectural changes necessary for current processors to support intrathreads. Section 3 provides examples of code transformations that can take advantage of intrathreads and evaluates their performance improvements. Section 4 discusses the related works. Finally, we give our conclusions in Section 5.

2 The Inthreads Architecture

An *intrathread*, or *inthread*, is a context of computation that participates in a control flow of a single OS thread. The processor executes the active inthreads concurrently by interleaving the issue of their instructions. There is no context switch between the inthreads.

Furthermore, inthreads share the registers in the register file. In this way, the shared registers can be used for communication, similarly to the way the shared memory is used in conventional concurrent programming, but with much better bandwidth and latency. This also offloads the traffic of inthread communication from the memory subsystem.

Inthreads feature a synchronization mechanism, consisting of a set of binary semaphores, which we call *condition registers*. Operations are defined to suspend an inthread execution until a certain condition register is set, and to manipulate the state (set/cleared) of condition registers.

In the suspended state, the inthread does not consume any computational resources of the processor. The processor avoids fetching its instruction until some other inthread sets the necessary condition register. Then, the processor resumes fetching and issuing the suspended thread's instructions and it joins the computation.

Since the inthreads use shared registers for communication, a memory consistency model must be defined that governs the register accesses of different inthreads. The model we use is Release Consistency [9]. This model defines Release and Acquire semantics for special operations. Informally, a Release operation ensures that the results of all the operations preceding it in the program order will be visible to any following operation. An Acquire operation en-

sures that all the read operations following it in the program order will see any update which precedes it.

In our case, starting an inthread and setting a condition register have the Release semantics, and waiting on the condition register has the Acquire semantics. Together, they ensure that if two threads synchronize through the condition registers mechanism, all the updates performed by one of them will be seen by the other one.

Note that inthreads do not replace other parallelization mechanisms, such as Simultaneous Multithreading [5] or HyperThreading [7], but rather complement them. These mechanisms enable concurrent execution of different OS threads, while inthreads parallelize the execution of a single OS thread.

The sharing of the register file and the absence of the context switch makes inthreads extremely lightweight, both in terms of the runtime overhead and of the implementation in the hardware.

The amount of information necessary for implementing an inthread context can be as low as one Instruction Pointer register. Practically, the amount of state necessary to achieve high performance of inthread instructions is higher, but still minor compared to the processor itself. Due to the size restrictions of this paper, we cannot discuss here the details of inthreads implementation.

Since inthreads do not introduce any new computational resources to the processor, they offer no benefit for code that is fully utilizing the processor. Instead, inthreads provide speedups by improving the processor utilization, up to its maximal capability. Therefore, a processor with inthreads capability can utilize more computational resources for the same computation. It can then make sense to include more functional units or other resources in such processor.

The current specification of inthreads mechanism has several important limitations. These limitations reduce applicability in certain situations and will be addressed as a part of the future research. In particular, we find that some features of the IA-64 [10] architecture can be used for solving those problems.

The first limitation is concerned with the allocation of inthreads. With the OS threads, the software can, using context switches, create any number of threads requested by an application. In the hardware, however, the number of inthreads that can be supported is limited.

While some algorithms need only a few inthreads and can adapt to any available amount, others may not be that flexible. The study of different techniques that utilize inthreads will provide the optimal number of inthreads that has to be supported.

In addition, as each inthread needs several dedicated registers to perform its computation, the practical number of inthreads is also limited by the size of the register file. Fortunately, modern architectures, such as IA-64, feature large register files.

Another limitation is that inthreads are not transparent with regard to function calls. In a regular processor, a function can assume that it has exclusive control over the register file. Therefore, if it needs to modify some register transparently to the caller function, it can save its contents in memory and then restore it before the return. With inthreads, the assumption is no longer true—other inthreads can be active that access the same registers.

This limitation can be addressed by rotating register windows in architectures like IA-64. When a function is called, a new frame is allocated for it from the register file, and it can safely assume that no caller is accessing these registers.

2.1 Extensions of the Instruction Set

The instructions added for inthreads support are `it.start`, `it.kill`, `it.halt`, `it.wait`, `it.cond.set` and `it.cond.clr`. A set of one-bit condition registers is defined, and used by `it.wait`, `it.cond.set` and `it.cond.clr`.

- `it.start TID,OFF` is a special kind of an unconditional branch instruction. It has two parameters: `TID` is the id of the thread to be created, `OFF` is the offset to the instruction from which the new inthread will begin execution. When receiving this instruction, the processor starts executing instructions in a context designated by `TID`. If the context was active prior to execution of `TID`, its activity is aborted.
- `it.kill TID` deactivates the context designated by `TID`. The corresponding inthread is effectively killed.
- `it.halt` is issued by an inthread to halt itself. This instruction deactivates the thread synchronously, allowing all the instructions already issued by it to complete. This instruction is used by an inthread to stop itself when it completes its task.
- `it.wait C` checks the contents of the condition register `C`. If the register is set, the instruction proceeds and the register is cleared. If the register is clear, the inthread that issued it is suspended until some other inthread sets the register. If several `it.wait` instructions try to access the same condition register at the same time,

only one of them will succeed, and the rest will be suspended.

- `it.cond.set C` and `it.cond.clr C` manipulate the contents of a condition register `C`. `it.cond.set` sets the value of the corresponding condition register. If a `it.wait` instruction of some inthread is suspended on register `C`, it is resumed and the register is cleared again. `it.cond.clr` instruction clears the corresponding condition register.

3 Code Transformations

In this section we show code examples and the transformations that can be applied to them in order to achieve parallelism. These examples show the wide scope of concurrent programming techniques that are applicable to inthreads.

The list of transformations presented here is by no means complete. More concurrent programming algorithms will be adopted to inthreads with time, and more code patterns that allow parallelization will be discovered. The study of these transformations is one of the major goals of future research.

To evaluate the performance improvements offered by the parallelizations, a simulator was implemented based on the SimpleScalar software suite [3]. Our simulator models a superscalar, fully out-of-order machine with a configurable number of instructions that can be issued during each cycle. The issue width of the processor is shared among the active inthreads. Due to space limitations, we cannot present more details on the simulator here.

All the code transformations presented here have been performed manually by changing the assembly code produced by the compiler. Automatic generation of inthread-parallel code is a target for future research.

Most of our examples consider optimization of a loop with certain control and data flow structure. For each one, we present the original code excerpt, the transformed code and simulation results that evaluate the potential speedup of the transformation.

The examples presented in this section are written in pseudocode, where plain C code is interleaved with the inthread-specific assembly commands. Note that the code cannot be trivially compiled from C. The reason is that inthreads running concurrently must take care to use different registers from the register file.

3.1 Worker Pool

Consider the following code:

```
for(int i=0; i<size; i++) {
    // perform the work on the chunk
}
```

If the iterations of a loop are completely independent of each other, we can parallelize the loop by creating a pool of worker inthreads to perform the work of different iterations in parallel. The main inthread T_0 runs the loop and produces the chunks of work to be performed. Each one of the worker inthreads $T_1 \dots T_n$ runs a loop in which it takes a chunk and executes it.

To make sure no two inthreads try to work on the same chunk, we use two condition registers C_1 and C_2 . The variable i is mapped to the same register for both T_0 and the worker inthreads. T_0 signals that the new value is available in the variable i by setting C_1 . Then, one of the worker threads that was waiting on C_1 awakes and writes the value to its own register. It then sets C_2 to signal T_0 that it can continue to the next value of i .

The following pseudo-code describes the loop performed by T_0 .

```
for(int i=0; i<size; i++) {
    it.cond.set C1
    it.wait C2
}
```

The following is the code performed by each one of the worker inthreads. Note that except for the shared variable i , all the rest of registers are different for all the workers.

```
for (;;) {
    it.wait C1
    chunk = i
    it.cond.set C2
    // perform the work on the chunk
}
```

A code of the corresponding structure can be found in the 184.crafty benchmark of SPEC2000. Figure 2 shows the speedup achieved by the worker pool optimization. The graph displays the speedup achieved by using different number of worker inthreads as it depends on the processor issue width. With one worker, the code is only marginally faster than the original because the bulk of work is not parallelized. With several workers, the maximal obtainable speedup is linear in the number of workers.

Note that the more workers are used, the higher is the issue width of the processor necessary to reach



Figure 2: RESULTS OF THE WORKER POOL PARALLELIZATION.

the maximal speedup. This indicates that the optimized version is able to utilize more of the resources offered by the processor. Note that at extremely high processor issue width the speedup is strictly linear in the number of workers.

3.2 Loop pipelining

Consider the following code sample:

```
for(int i=0;i<size;i++) {
    for(int j=0;j<size2;j++)
        // perform work
}
```

In some cases, it can be found that the computation of iteration i, j of the inner loop is dependent only on the iterations $1 \dots j - 1$ of the previous iterations of the outer loop.

Then, we can split the inner loop into n blocks of fixed number of iterations. n worker inthreads will be allocated to perform the work, each one of its corresponding block. Each worker runs the same outer loop, but in the inner loop the k th worker performs only the k th block.

To keep the parallelized computation equivalent to the original one, the k th worker must not begin processing the i th iteration until the $k - 1$ -th worker completes its i th iteration. To achieve this, we use two condition registers to perform the handshake between the k th and $k + 1$ -th worker.

The following code outlines the implementation of worker T_0 .

```
for(int i=0;i<size;i++){
    for(int j=0;j<block0;j++) {
        // perform work
        it.cond.set C1
        it.wait C2
    }
}
```

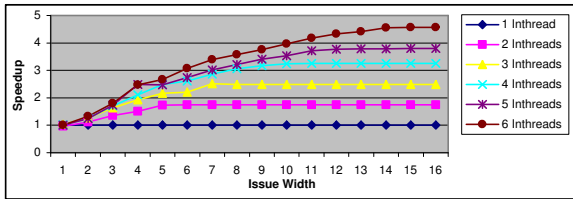


Figure 3: SPEEDUP GAINED BY LOOP PIPELINING.

The rest of the workers must handshake with both the previous worker and the next one. The following code outlines the implementation of worker T_1 .

```
for(int i=0;i<size;i++){
  for(int j=block0;j<block1;j++) {
    it.wait C1
    it.cond.set C2
    // perform work
    it.cond.set C3
    it.wait C4
  }
}
```

For simplicity, we have omitted the code that handles the setup and termination of the worker inthreads.

The performance of the loop pipelining transformation is very sensitive to the work distribution among the worker threads. The time required to perform the parallelized loop is at least as long as the time of any of the worker threads to perform its part of the work. If the time is not divided perfectly equally, the speedup will be sub-linear.

A loop of such structure can be found, for instance, in the 256.bzip application of the SPEC2000 benchmark suite. Figure 3 shows the result of parallelizing the loop by splitting it to a different number of chunks.

3.3 Loop partitioning

In a more general case, each iteration of a loop can consist of several blocks of varying size. Each one of the blocks can include complex branching patterns and have a varied-length loop by itself.

To parallelize such loops, we can use various combinations of loop pipelining and work pooling. In some cases it may even be beneficial to group several iterations of a loop into a unit of work, and then execute the units of work on the pool.

To illustrate loop partitioning, consider again the code in Figure 1. As stated before, the traversal of the linked list can be executed in parallel with the `strlen` calls. Moreover, different `strlen` calls are independent from each other, therefore we can use

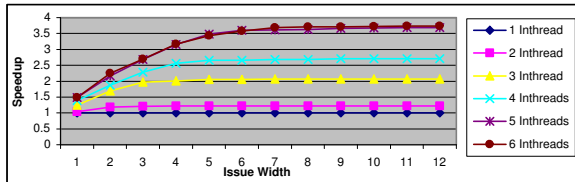


Figure 4: RESULTS OF LOOP PARTITIONING.

several worker inthreads to execute the calls concurrently.

Figure 4 shows the results of optimizing of the code. The version with two inthreads simply uses an additional inthread for the `strlen` calls. Further speedup is obtained by using several worker threads to perform the string length calculations. With five worker inthreads, one for the linked list traversal and four additional workers, the maximal possible speedup of about 3.7 is reached at the point when the traversal becomes the computation bottleneck.

3.4 Speculative Computation

Speculative execution of instructions is one of the more advanced techniques for improving instruction latency tolerance in modern processors. With inthreads, entire blocks of code can be executed speculatively.

Consider two consecutive blocks of code B_1 and B_2 , where the decision to perform B_2 depends on the result of executing B_1 . B_2 can be spawned in a new inthread T_1 which will start executing immediately. When B_1 is completed, then, depending on the outcome of the computation, it either waits for the result of B_2 or kills B_2 and continues.

To illustrate this technique, consider the following loop. Two strings are compared to find the first place where they do not match.

```
while (*scan && *++scan == *++match);
```

To parallelize the loop, we simply split it. First, several iterations are performed in T_0 . The rest of the iterations are performed speculatively in the worker thread T_1 .

Figure 5 shows the result of parallelizing the function with varying number of iterations in T_0 . Five different input sets have been generated for the function. In the first input set, all the strings have common 4-character prefix. In the second, all strings share the same 3-character prefix, and there are 3 subsets, where all strings in each subset have a common 4-character prefix. The third input set has a common 2-character prefix with 3 subsets sharing a common

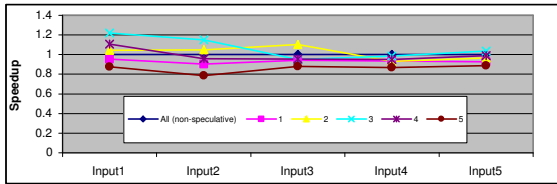


Figure 5: RESULTS OF SPECULATIVE EXECUTION. THE GRAPHS CORRESPOND TO THE NUMBER OF ITERATIONS PERFORMED NON-SPECULATIVELY BY T_0 .

3-character prefix, etc. Every string was compared to every other string, and the average speedup was measured on each input set.

Observe that running only one iteration in T_0 will always degrade the performance: because of the latency of starting an inthread, T_1 will not be able to run in parallel with T_0 . Running 5 and more iterations in T_0 will degrade the performance too, since the results of T_1 are almost never used. However, in cases where the execution of T_0 is balanced with that of T_1 , we can see the speedup. The maximal speedup is 22%.

This example shows that inthreads can be applied at extremely low granularity. It also demonstrates that speculative execution must be used with great care. Because speculative execution involves performing extra work which may be discarded, it may sometimes lead to slowdowns.

To optimize the loop above effectively, the compiler must be able to infer the average number of the iterations that the loop will perform. This can be achieved with profile-guided optimization.

3.5 Other code transformations

We believe that the inthreads architecture is simple and general enough to implement many of the algorithms and patterns established in the field of concurrent programming. In addition, totally new techniques can be developed that utilize the low level of granularity offered by inthreads.

For example, consider a program that traverses a search tree (or performs binary search on an array).

Normally, we compare a node n with the key, and then descend to either $n \rightarrow \text{left}$ or $n \rightarrow \text{right}$, depending on the outcome. With inthreads, we can run comparisons of $n \rightarrow \text{left}$ and $n \rightarrow \text{right}$ speculatively in parallel to comparing n . When the comparison of n finishes, we can decide on which of the speculative executions was indeed the necessary one and spawn two new inthreads to compute its sons speculatively. The unnecessary speculative inthread is killed.

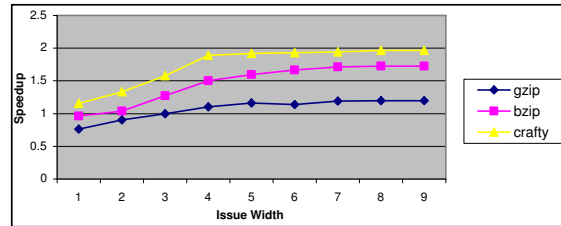


Figure 6: RESULTS OF PARALLELIZATION IN SPEC BENCHMARKS.

Although one third of the work performed is wasted, three inthreads are working all the time. Thus, provided that the processor is wide enough, the total speedup of the code could reach up to 2.

3.6 SPEC Benchmarks

As noted above, inthreads are beneficial only for code that is otherwise unable to fully utilize the processor resources. It is therefore important to evaluate which fraction of time in standard programs can be parallelized with inthreads.

To this end, we have explored several programs of the SPEC2000 benchmark suite: 164.gzip, 184.crafty and 256.gzip.

Since the compiler have not been implemented yet, we applied the inthreading optimization only to a single function in each benchmark and measured the speedup of that function. In each case, we have profiled the program and selected for optimization the function with the highest percentage of the running time. In 164.gzip the selected function takes about 40% of the execution time, and in 184.crafty and 256.bzip the selected function takes about 20% of the execution time.

For each function, one or more transformations described in 3.1, 3.2, 3.3 and 3.3 have been applied. The speedup factor ranges from about 1.2 for 164.gzip to 2.0 for 184.crafty, which translates to up to 10% speedup for the whole program. If the whole program was optimized, we could expect better overall speedups.

Figure 6 displays the results of optimization of all the functions.

4 Related Work

The ideas implemented in the IA-64 architecture [10] solve the problems of control and data dependencies to some extent by using predicated execution and speculative memory access instructions. These mechanisms allow one to fold several paths of branch

instructions into a single serial stream of computation that can be efficiently parallelized. However, there are many cases in which the techniques employed in the IA-64 do not provide a solution. For instance, predicated execution cannot handle jamming together two independent loops of potentially different lengths.

Simultaneous Multithreading [5] and its predecessors [13, 12] provide mechanisms for sharing processor execution resources between several threads to maximize resource utilization. Single Chip Multiprocessors [2] allow to build many processors on a single chip, sharing many of the execution resources.

Most modern processors implement these techniques to some degree: Alpha EV8, HP PA-RISC 8900, IBM Power4, Intel HyperThreading, AMD SledgeHammer. These architectures are similar to Intrathreads in that they increase utilization of the processor’s resources by sharing them among several threads. However, the only communication and synchronization mechanism available is still the main memory. While it can be reasonably efficient for synchronizing large chunks of computation, it is prohibitively expensive for the kind of low granularity communication that intrathreads need.

Many works attempt to provide necessary communication and synchronization mechanisms to parallelize the computation.

Tullsen et. al. [14] describe a low overhead synchronization mechanism for SMT processors. Though similar to inthreads, this work opts for memory-based locks and communication. This approach provides for a good scalability, but requires more OS involvement to manage. The main difference of inthreads is the communication between threads through architectural registers. While having its own limitations, our mechanism allows for the algorithms to be specified at lower levels.

Micro-threads [4] are similar to inthreads in that they share the register file between different threads and provide data synchronization. However, several differences make them unsuitable for most of the transformations presented in this paper. Most important, micro-threads synchronize on the data registers only and lack therefore the synchronization semantics necessary for complex parallelization patterns.

Multiscalar Processors [8] support grouping of instructions into coarse-grain tasks. The processor can then execute the tasks in parallel, handling dependencies between them similarly to the way it handles dependencies between individual instructions. With Intrathreads, there is no division into tasks visible to the processors. Instead, explicit synchronization in-

structions are used at software level, providing better flexibility and more opportunities for parallelization.

Supertreaded architecture [11] supports a special fork instruction that spawns a new thread of execution. The register file is copied on fork, allowing for efficient data transfer to the spawned thread. Each given thread can spawn an additional one only once, thus converting the execution of the program into a virtual pipeline. In contrast, Intrathreads feature free communication between the threads, allowing for applicability in larger number of cases. In addition, Intrathreads share a single register file, allowing for a much simpler implementation.

The Weld architecture [6] spawns threads during the program execution to tolerate long-latency instructions. All spawned threads are executed speculatively and merged later into the main computation. The architecture defines an additional fork instruction that is a combination of branch with a fork. Upon executing this instruction, if an execution context is available, the processor spawns a new thread at the new address. This technique improves the parallelism only to a certain degree. Indeed, the authors conclude that there is no significant benefit of using more than two execution contexts per processor.

The Tera machine [1] can run massive numbers of in-core threads simultaneously. The architecture defines lightweight memory-based locking mechanisms for thread synchronization. Unfortunately, programming the machine requires the programmer to explicitly parallelize the computation into a high number of threads.

5 Conclusion

In many cases of an “inherently sequential” code, it turns out that the parallelism is just too low-level to be expressed by means of concurrent programming with threads, but too high-level to implement in terms of a regular processor architecture. In such cases, inthreads provide the middle layer between the instruction-level parallelism and the thread-level one.

We have described the instruction set necessary to implement inthreads and shown some of the code transformations that use concurrent programming algorithms and patterns to parallelise pieces of sequential code.

We have applied these transformations to some synthetic benchmarks and also have shown that the opportunities for these optimizations can be found in some important parts of code in the SPEC benchmark suite. We have shown that such transformations can reach speedups almost linear in the degree

of parallelization. However, inthreads are beneficial only for code with low instruction-level parallelism. Therefore, it remains to be seen how much speedup can be gained by inthreads when applied to whole programs.

A lot of research is still needed in the study of inthreaded architectures. In particular, there is a great challenge in the integration of compiler transformations that will automatically produce inthreaded code.

References

- [1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6, Jun 1990.
- [2] B. A. Nayfeh, L. Hammond and K. Olukotun. Evaluation of Design Alternatives for a Multiprocessor Microprocessor. In *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996.
- [3] D. Burger, T. M Austin, and S. Bennett. Evaluating future microprocessors: The SimpleScalar tool set. Technical Report CS-TR-1996-1308, University of Wisconsin-Madison, 1996.
- [4] C. Jesshope and B. Luo. Micro-threading: A new approach to future RISC. In *ACAC*, pages 34–41, 2000.
- [5] D. M. Tullsen and S. Eggers and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, 1995.
- [6] E. Özer and T. M. Conte and S. Sharma. Weld: A Multithreading Technique Towards Latency-Tolerant VLIW Processors. *Lecture Notes in Computer Science*, 2228:192–202, 2001.
- [7] D. T. Marr et. al. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6:4–15, Feb 2002.
- [8] G. S. Sohi and S. E. Breach and T. N. Vijaykumar. Multiscalar Processors. In *25 Years ISCA: Retrospectives and Reprints*, pages 521–532, 1998.
- [9] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. 17th Ann. Intl. Symp. on Computer Architecture*, pages 15–26, May 1990.
- [10] <http://developer.intel.com/design/itanium/manuals/index.htm>. The IA-64 Architecture Manual.
- [11] J.-Y. Tsai and P.-C. Yew. The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation. In *Proceedings of Int'l Conf. on Parallel Architecture and Compiler Techniques*, pages 35–46, 1996.
- [12] R.G. Prasad and C.-L. Wu. A Benchmark Evaluation of a Multithreaded RISC Processor Architecture. In *International Conference on Parallel Processing*, pages 84–91, 1991.
- [13] S. W. Keckler and W. J. Dally. Processor coupling: integrating compile time and run-time scheduling for parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 202–213, Gold Coast, Australia, 1992.
- [14] Dean M. Tullsen, Jack L. Lo, Susan J. Eggers, and Henry M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Fifth Annual International Symposium on High-Performance Computer Architecture*, pages 54–58, Jan 1999.