A Comparison of Different Multithreading Architectures

Jeroen Hordijk Henk Corporaal

Delft University of Technology
Faculty of Information Technology and Systems
Section Computer Architecture and Digital Systems
Mekelweg 4, 2628 CD Delft, The Netherlands
e-mail: {J.Hordijk,H.Corporaal}@its.tudelft.nl

Keywords: parallel computing model, multithreading architectures, compiler optimizations for multithreading

Abstract

A new direction in the research of automatic extraction of parallelism from sequential high level languages focuses on the concurrent execution of multiple threads, *i.e.* multithreaded architectures. The interest for architectures that are able to follow multiple control paths is stimulated by reaching the border of available parallelism for single thread architectures. On the other hand, there is a large potential for performance improvement using multithreaded architectures. And future hardware technologies promise such large number of gates, so that duplication of complete processors on a single chip can be achieved. This paper classifies and compares the different theoretical approaches and practical implementations of several existing multithreaded architectures.

1 Introduction

Most of the research into the exploitation of parallelism from sequential high level languages has been focussed on the execution of a single control flow. Several studies for general purpose applications have shown that the maximum exploitable parallelism is in the range of 5 to 10 [1, 2] for a practical implementation of a single speculative processor. As the technology progresses future architectures will be able to exploit more parallelism than is available from single control flow execution. A new potential for speedup lies in the simultaneously execution of multiple flows of control [1, 3]. Executing multiple control flows means splitting the instruction window over multiple execution units. Therefore executing multiple control flows allows to increase the overall instruction window. Increasing the instruction window in processors with a single execution unit is difficult, because the hardware complexity needed to check the dependencies increases with the square of the number of instructions in the window.

There is no consensus on the naming for processors that execute multiple flows of control. In this paper we will use the term multithreading architectures for architectures, which are able to execute multiple dependent tasks in parallel. Note that, roughly two other classes of multithreaded architectures are known. First, architectures with only one execution unit but special hardware to support fast thread switching. Second, architectures that execute multiple independent threads concurrently. Mostly these architectures mix the instructions of several different threads on one execution unit, also called simultaneously multithreading.

The essential aspects of multithreading architectures concentrate on the way the interthread dependencies are handled. When the architecture has no hardware support for resolution of interthread dependencies only completely independent threads can be executed concurrently. The programs written in imperative languages usually contain many dependencies, therefore this approach will lead to inferior performance gains. The key to make multithreading work is to develop both compiler and hardware techniques for efficient dependence resolution.

The goal of this paper is to gain insight into advantages and disadvantages of different architecture approaches. We will study the theoretical possibilities and investigate the different implementations in order to achieve this goal. In section 2 the paper explores the design space of these techniques in order to investigate the future directions of the research into multithreaded architectures. In the third section more detailed study of the implementation of several existing multithreaded architectures follows. The fourth section analyzes the operation of these architectures using several small benchmarks. The paper ends with the conclusions.

2 Classification

In order to study the design space of multithreaded architectures and accompanying compilers, we present a classification of these processors on the most essential characteristics; this classification is shown in table 1. The system characteristics are divided into five main parameter classes, each of these classes is further divided into several specific parameters for which the table gives the possible values. The classification focuses on the characteristics of the software code of threads, the scheduling of threads and the handling of dependencies between threads. The classification contains a wide range of architectures from very conservative to extreme aggressive. For example, a conservative architecture would not allow concurrent execution of dependent threads and has no support for dynamic dependence analysis. In this case the compiler should then avoid all dependencies between threads that are executed in parallel. Just like a conservative architecture, the compiler of an aggressive architecture tries to determine the dependencies as accurate as possible. However, it has also hardware support for dynamic dependence analysis, which gives more precise dependence information. Furthermore, the architecture will allow speculation in case the dependencies are not known.

Partitioning: We assume that the source code is written in a sequential language, so that the code must be divided into threads that run on the different execution units. Partitioning can be done by the compiler and in theory also at run-time by the hardware. However, partitioning at run-time is rather expensive, because all instructions must be loaded and decoded by a central unit first. Another consequence of dynamic partitioning is that both scheduling and dependence analysis must also be done at run-time. Therefore partitioning is mostly done at compile-time.

The partitioning granularity is a critical parameter. Multithreaded architectures exploit parallelism on two levels. Within an execution unit fine grain parallelism (on instruction level) is exploited, between the different units more coarse grain parallelism must be exploited. Choosing the granularity too small will be inefficient because too much time is spend on spawning and retiring of threads, and too little instruction level parallelism is left for the execution units to exploit. On the other hand, chosing the granularity too large will lead to little thread parallelism, so that the processing units will not efficiently be used.

When the threads are dependent there must be some sort of hardware to handle these dependencies in order to execute these threads in parallel. The partitioning algorithm can be instructed to exclude certain dependencies because the architecture has no hardware support to resolve these dependencies at run-time. These dependencies must be resolved at compile-time and therefore potentially less parallelism can be exploited. Sometimes the source code can be rewritten, for instance register dependencies can easily be converted to memory dependencies. This may be useful when run-time memory dependence checking is supported, but no inter-processor register communication is available. The other way around is difficult because the behavior of the memory dependencies may not be completely known at compile-time. When control dependencies are not allowed, it means that all control paths from the 'forking' thread should always reach the 'forked threads'.

Thread Management: Thread management deals with the way threads are started on the different execution units and in which order. The scheduling algorithm decides when a new thread is started, this can be done both at compile-time or run-time. In the first case, in the source code the information about the successor thread or threads is included. Dynamic scheduling is only useful when also control speculation is allowed. When threads are control dependent and the architecture speculates on these dependencies, doing scheduling at run-time allows dynamic prediction schemes that are usually more accurate than static prediction techniques. Thread ordering differs also for systems with a single successor and multiple successor threads. In case a thread has multiple successor threads the main thread will not die but continue executing, as opposed to single successor execution[1]. The single successor variant combines the best with partitioning granularity of basic blocks and loops, because of the sequential behavior on this program level. Whereas, the multiple successors variant is more suitable for partitioning on the level of procedures.

Another possibility is to include all dependence information between threads in source code. The hardware can decide which threads can start executing based on this dependence information and knowledge of already resolved dependencies. This is *dataflow* kind of thread management, which is always combined with dynamic scheduling.

Dependence Analysis: The dependence analysis of register and memory dependencies is done separately, because of the different nature of register and memory dependencies. The control flow of the code within a thread causes that the register dependencies behave dynamically. Therefore static register dependence analysis must be conservative, and dynamic register dependence analysis can be more accurate.

Although the previous discussion also holds for the analysis of memory dependencies, the compile-time memory analysis has to deal with another more severe problem [4]. References to memory locations do not have to be absolute, but can be an expression that

[1]Note that, strictly speaking the combination of multiple successors and dying parent thread is also possible.

Parameter Class	Parameter		Range
Partitioning	partitioning time	par-tm	[static, dynamic]
	partitioning granularity	par-grn	[instruction, basic block, loop, procedure]
	allowed thread dependencies	trd-dep	[register, memory, control]
Thread Management	scheduling time	schd-tm	[static, dynamic]
	thread creation	trd-crt	[single successor, multiple successors, dataflow]
Dependence Analysis	register	da-reg	[static, dynamic]
	memory	da-mem	[static, dynamic, both]
	control	da-ctrl	[static, dynamic]
Synchronization	register release time	reg-rls	[late, early]
	register strictness	reg-stc	[strict, non-strict]
	memory release time	mem-rls	[late, early]
	memory strictness	mem-stc	[strict, non-strict]
Speculation	register	spc-reg	[yes or no]
	memory	spc-mem	[no, load, store, both]
	control	spc-ctrl	[no, single path, multiple paths]

Table 1: The classification of multithreaded architectures.

must be evaluated run-time. Therefore, memory dependence analysis is more complex and this usually leads to highly conservative compile-time estimation of the actual (run-time) dependencies.

Synchronization: Data communication between threads can be viewed from the two sides: definition of data that resulted in the dependence and the use of this data. There are two possible times to forward the data values on which other threads are dependent. One is directly after the data is produced, which we will call *early release* and the other is when the thread is finished, named *late release*. Naturally, the first scheme results in the best performance, but at the cost of more complex hardware synchronization support.

Where the release time is related to the definition of data values, the *strictness* of the execution model relates to the use of the data values. Just like the release time there are two variants of strictness. Either the scheduling algorithm requires that all input dependencies are satisfied before a thread can start, this is called *strict execution*. Data dependencies can also be resolved during the execution of the thread, *i.e.* threads are allowed to start executing before all dependent data values are available, which is called *non-strict execution*.

Speculation: In principle speculation can be useful for all dependencies that have a dynamic behavior or for dependencies of which the static behavior is not known at compile-time. These dependencies can be of all three types: register, memory and control dependencies.

There are three variants of the speculative memory operation feature: only speculative loads, only speculative stores and the combination of speculative loads and speculative stores. In order to limit the complexity of speculative memory operations, one can allow only speculative loads. This saves the store buffer hardware needed for speculative stores. Memory stores are destructive and can not be repaired after a misprediction, so that hardware support is needed. The standard solution is a store buffer that holds the stores until the prediction is evaluated. Theoretically, also only speculative stores can also be allowed, but this is not interesting from the performance point-of-view.

Speculation of control dependencies has two main variants. One can try to predict the single path through the control flow graph that the program will follow during execution. Another possibility is to execute multiple paths and squash the not taken path(s) latter on. Of course, the latter has higher performance potential, but also at the cost of wasting many resources. Speculation can be done at both compile-time and run-time. When it is done at run-time dynamic dependence analysis is required. On the other hand, the advantage of run-time speculation is that a dynamic prediction algorithm can be used, which have higher prediction hit rates.

3 Architectures

A few multithreading architectures that are able to handle dependent threads are proposed yet. In this section those architectures are discussed and grouped using the classification of the previous section. This will give an overview of which areas from the design space are studied and which are not. We will not go into all implementation details of the discussed architectures, see [5] for a more detailed analysis.

Multiscalar Architecture: In table 2 the classification of the Multiscalar architecture is given [6]. Threads are generated by the compiler and contain mostly a single loop iteration. The architecture allows all thread partitionings as long as threads have a single successor. The compiler 'blindly' partitions the program in threads, so that the threads can contain all possible dependencies, *i.e.* register, memory and control dependencies.

The Multiscalar hardware contains a sequencer unit that determines the next to be executed thread. The possible successors of a thread are summarized in the

Parameter	Multiscalar	Superthreaded	SPSM	Parafrase
par-tm	static	static	partial dynamic	static
par-grn	loop	loop	basic block	multilevel
trd-dep	memory, memory, control	memory, control	(register), memory	register, memory, control
schd-tm	dynamic	static	static	dynamic
trd-crt	single successors	single successor	multiple successors	dataflow
da-reg	static	static	static	static
da-mem	dynamic	dynamic	dynamic	static
da-crtl				
reg-rls	early	-	-	early
reg-stc	non-strict	-	-	strict
mem-rls	early	early	early	early
mem-stc	non-strict	non-strict	non-strict	strict
spc-reg	no	-	-	no
spc-mem	load and store	no	load	no
spc-ctrl	single path	single path	-	no

Table 2: The classification of several existing multithreading architectures.

thread descriptor, which is inserted in the source code by the compiler. The dynamic scheduling allows for dynamic speculation of control dependencies.

The processing units are connected with an unidirectional ring, which combines well with the single successor nature of threads. As long as there are idle processing units the sequencer unit speculatively starts new threads on these units. This way oldest running thread, which is called the tail or main thread, contains always the correct processor state. The other threads are all speculative and may need to be squashed.

The register dependencies between threads are handled by the compiler. Read-after-write dependencies between threads are located, and the last register writes are tagged with a bit that the hardware notifies that the register value must be send to the successor thread. In combination with control flow within a thread a unique last register write may not exist, in that case an explicit release instruction is used. Both write-after-read and write-after-write dependencies cause no problem because all processing units have separate register files.

Memory dependencies are completely handled at run-time by a unit called the *Address Resolution Buffer (ARB)*. It stores all memory operations, *i.e.* both load and stores, the addresses and the processing unit numbers. Using this information the unit can verify whether all memory operations are done in the correct order (*i.e.* whether the dependencies are obeyed). The table holds all the data values of all stores, therefore the buffer can also resolve write-after-read and write-after-write memory dependencies.

Superthreaded Architecture: The Superthreaded architecture classification can also be found in table 2 [7]. The processing units are connected in an unidirectional ring, just like in the Multiscalar architecture, and threads have a single successor. In contrast with the Multiscalar architecture, which executes threads highly speculative, the Superthreaded architecture analyzes dependencies before they occur. Another difference is the static scheduling of the Superthreaded architecture.

Register dependencies between threads are completely avoided by the compiler. Memory dependencies are completely obeyed (no speculation is done) by a combined software/hardware technique. The compiler rewrites the program code so that all addresses of memory writes are calculated in the beginning of the thread. Future threads do not start executing before all addresses are known. A special hardware buffer keeps track of all addresses. In this way memory dependencies are perfectly aligned, *i.e.* dependent memory reads can start directly after the store and independent memory reads do not have to wait. On the other hand, it may take a lot of calculations before the write addresses are known. In this time successor threads may be idle. This memory dependence analysis technique forces some kind of thread pipelining.

Single Program Speculative Multithreading Architecture (SPSM): The major difference between the SPSM architecture and the two previous architectures is the scheduling of threads over the processing units [8]. The Multiscalar and Superthreaded architecture have a strictly sequential order of executing threads whereas in the SPSM architecture a thread can have multiple successors. The program code contains thread fork instructions that can be optionally honored by the hardware. When a successor thread finishes its results are copied into the main thread. The hardware has no support for run-time handling of register dependencies, only for memory dependencies. Control dependencies can be analyzed dynamically but no speculation is allowed. Although the architecture can not pass register values during the execution of a thread, threads may be register dependent. Only a single control path inside the thread needs to be register independent with the forked thread. To make sure the right path(s) is followed the compiler inserts suspend instructions that indicate the path(s) that need to-be-followed. The hardware is responsible for checking that one of these paths is actually taken. Otherwise, the thread is squashed.

Parafrase Architecture: The Parafrase project studies the exploitation of functional parallelism using the Hierarchical Task Graph (HTG) [9]. The HTG is an intermediate program representation, which is the basis for code generation and program optimization. The thread graph is structured such that it fully encapsulates the parallelism of all types and at all levels of granularity. All dependencies between the threads are analyzed by the compiler. Because of the limitations of compile-time analysis techniques, the compiler must conservatively assume a lot of non-existing dependencies. Special hardware collects the resolved dependencies. When all dependencies for a task are satisfied the task is started, *i.e.* a combination of strict execution and dataflow kind of thread management. Note that all other described architectures allow tasks to start when not all dependencies are resolved, *i.e. non-strict execution*.

4 Architecture Execution Examples

In this section we will investigate the operation of different architectures for different benchmarks. This will give more insight into the operation of the architectures. For instance, about the kind of code pieces that can be executed efficiently and about code constructions that cause problems. The benchmarks are mainly general-purpose programs written in C.

4.1 Linked List

The first example code is taken from [6], see figure 1 for the C code. The execution repeatedly takes a symbol from a buffer and runs down a linked list checking for a match of the symbol. If a match is found, a function is called to process the symbol. If no match is found, an entry in the list is allocated for the new symbol. After an initial startup, additions to the list become infrequent, because most symbols match an element already in the list. We make the following assumptions about the functions: the `addlist` inserts a new symbol in the linked list at a place that can be anywhere in the list. The function `process` keeps track of the number of times a symbol has occurred in the buffer. This makes the calls to `process` possibly dependent.

This benchmark contains a nested loop, and either the inner loop or the outer loop can be partitioned. The iterations of the inner loop are dependent because they walk down a linked list. The iterations of the outer loop are only dependent when a symbol in the buffer occurs that is not yet present in the linked list. Of course, the frequency of this is highly dependent on the contents of the buffer array. Probably the chance of dependent outer loop iterations decreases fast as more symbols are inserted in the linked list.

```
for (index = 0; index < bufsize; index++) {
        symbol = buffer[index];
        for (list = listhead; list; list = lnext[list]) {
                if (symbol == lelem[list]) {
                        process(list);
                        break; }}
        if (!list)
                addlist(symbol); }
```

Figure 1: The C code for the linked list program.

Multiscalar Architecture: Most natural partitioning for the Multiscalar Architecture is to schedule individual loop iterations of the outermost loop on different processing units. This results in a register dependence on variable `index`. The counter is updated at the end of loop, whereas moving this calculation to the beginning of the loop will highly enlarge the amount of exploitable task parallelism. We have used a standard compiler, which does not take into account that the code is targeted for multithreaded architectures. And the given partitioning results in memory dependencies on the following variables. `listhead`: the new thread reads this variable in the beginning and it may be changed by previous threads. For example, when the first element is inserted into the (empty) list. `lnext`: when a new element is inserted into the list, it will create a dependence with future threads. If the new element is inserted in the beginning of the list the dependence will be on `listhead`, otherwise it will be on `lnext`. `lemen`: although there is a dependence it will never be activated because the dependence on `lnext` always becomes first. The threads are control dependent on the branch that tests whether the end of the buffer is reached. There are several different situations for the dependencies between two threads; they are shown in table 3. In each of the three cases different synchronization takes place, as described below: In the first case there are no memory dependencies, *i.e.* the symbol is already in the list and therefore the list is not changed. The control dependence between the threads can be speculated in this example with high accuracy, when the buffer of symbols is quite large. The branch is predicted to be taken, because this is the most taken direction. When the branch is correctly predicted the threads could be almost completely overlapped. A mispredicted branch (*i.e.* only happens after the last iteration of the outer loop) leads to squashing all future threads. The second and third case also contain memory dependencies on top of the already explained register and control dependencies. The variable `listhead` is read in begin of the outer loop, but this does not mean that the dependence is detected early. The Multiscalar proces-

dependencies		
register	memory	control
`r20`	-	yes
`r20`	`listhead lelem, lnext`	yes
`r20`	`lelem, lnext`	yes

Table 3: The possible dependencies between threads.

sor speculatively assumes there are no memory dependencies between threads. A thread follows the whole linked list before noticing that the symbol is not yet available in the list, and inserts it. Therefore future threads need to be squashed completely. In case a new symbol is inserted in the list close to the end, a lot of useful work is thrown away.

Superthreaded Architecture: The same partitioning as chosen for the Multiscalar architecture results in almost no speedup for the Superthreaded architecture. The Superthreaded architecture does not allow speculation on memory operations, whether a memory dependence is present is only known at the end of a thread. Therefore no significant overlap of threads is possible. In the Superthreaded architecture all registers that are changed in a thread and live over the thread border need to be passed in memory locations, because the architecture does not allow register communication. Therefore, the variables `index` and `symbol`, which were previously only stored in registers, must be saved to memory.

SPSM Architecture: The SPSM architecture does not allow speculation of control dependencies. We assume partitioning of the outer loop again. Therefore the next thread can only be started after the test `(index < bufsize)` is performed. Not only the update of the index variable must be done in the beginning of the loop like for the other architectures, but also the loop test. Otherwise no significant parallelism can be exploited.

Although the SPSM architecture has run-time support for register dependence analysis, it does not allow on-line register communication between threads. Therefore the previous proposed partitioning is not allowed without changes. One solution is to convert the register dependencies to memory dependencies. Both discussed adaptations will decrease the performance of the SPSM architecture compared to the Multiscalar.

Parafrase Architecture: The compiler of Parafrase architecture can partition the program in four hierarchical layers, *i.e.* procedure, loop, basic block and instruction. Analysing all dependencies on the basic block level results in more than twenty dependencies, so that the chance of an efficient partitioning is minimal. The outer loop is the most suitable candidate, when partitioning at the loop level. The same partitioning is chosen as for the Multiscalar architecture, so that also the same dependencies are present. However the architecture does not allow any run-time dependence analysis, therefore the compiler must do all analysis. The architecture will assume memory dependencies on variable `listhead`, `lnext` and `lemen` in each loop iteration, resulting in hardly any speedup.

```
int try (int i) {
    while (j != 8) {
        j += 1;
        if (b[j] && a[i+j] && c[7+i-j]) {
            x[i] = j;
            b[j] = false;
            a[i+j] = false;
            c[7+i-j] = false;
            if (i < 8) {
                if (! try(i+1)) {
                    b[j] = true;
                    a[i+j] = true;
                    c[7+i-j] = true; }
                else return (true); }
            else return (true); }}}
```

Figure 2: The C code for the queen program.

4.2 Queen's Problem

The benchmark solves the queen's problem for a square 8 board using backtracking (see figure 2). The program tries to place a queen at a horizontal line starting at the first line and then repeatedly continuing at the next lines. The array `x` holds the position of the queens; array `b`, `a` and `c` store taken lines in the vertical and both diagonal directions respectively.

The main procedure `try` of the queen's problem is recursive. This complicates the partitioning algorithm. The loop can not be partitioned without rewriting because it does not fit the sequential model demanded by the Multiscalar and Superthreaded architectures. These architectures only support the single successor thread creation technique, see section 2. However, a single loop iteration has multiple successors, because inside the loop the procedure `try` is recursively called. One way to solve this is to partition the loop only for the first recursive step (*i.e.* `i = 1`). This modification is fairly easy for static scheduled architectures like the Superthreaded processor, but somewhat more difficult for dynamically scheduled architecture like the Multiscalar.

Multiscalar Architecture: We will partition the different loop iterations of the first recursive step, *i.e.* `i = 1`, into threads. This requires quite sophisticated code transformation dependence analysis. First the recursive nature of the loop must be removed otherwise no partitioning is allowed. Copying the procedure `try` does this; the copied procedure contains the loop that will be partitioned and a call to the original procedure. The second procedure is still recursive, but will not be partitioned. Second constant propagation for `i = 1` is performed. This is not necessary but optimizes the code.

After that a thread contains the calculation of the queen's problem assuming that the position on the first row is already decided. Up to the maximum of 8 threads are dynamically created, one for each position of the first row. The number of spawned threads may be less because a solution is found earlier[2]. Three register dependencies are present between the threads: the register that holds the variable `j` and the register that holds the addresses of variable `a[j]` and `b[j]`. All three registers are updated in the beginning of the loop, so that parallelism can be fully exploited. The arrays `x`, `a`, `b` and `c` are (re)used in each thread, therefore a lot of write-after-read and write-after-write dependencies between the threads raise. The Multiscalar architecture is able to resolve these dependencies dynamically, as explained before.

Superthreaded Architecture: Choosing the same partitioning as we did for the Multiscalar architecture results in low performance because of the write-after-read and write-after-write dependencies between the threads. Unlike the Multiscalar architecture the Superthreaded architecture has no support to remove these dependencies; the operations that cause these dependencies must be placed in sequential order. Duplication of the arrays solves this problem.

SPSM Architecture: The SPSM architecture supports multiple successor threads therefore the loop in the queen benchmark can be partitioned without the need for code transformations. The instructions that fork new threads are skipped when all processing units are busy. This feature is essential for the execution of this benchmark, otherwise a deadlock will occur for architectures with less than eight processing units.

The partitioning suggested for the Multiscalar architecture and Superthreaded architecture is far more effective. Whereas the previous partitioning only required communication in the startup phase, in this partitioning the execution of the threads is almost completely sequential because of communication between the threads.

Parafrase Architecture: The execution of the queen benchmark on the Parafrase architecture is almost the same as for the SPSM architecture, when we assume partitioning on the procedure level. The dependencies between the threads force almost complete sequential execution. Just like the SPSM architecture the code transformation (procedure duplication) is not necessary but leads to far better performance.

5 Conclusions

We analyzed the design space for multithreaded architectures by summarizing the main characteristics of these architectures. The four discussed multithreaded architectures mainly differ in whether their behavior is more static or more dynamic. The Multiscalar architecture can handle the most dependencies at run-time, so that almost all loops with inherent parallelism can be effectively exploited. However, the hardware structure of the architecture forces threads to have a single successor. The Superthreaded architecture does not allow memory speculation; this prevents the architecture from exploiting parallelism where infrequent memory dependencies between the threads occur. The architecture is targeted to exploitation of loop level parallelism, just like the Multiscalar. The Single Program Speculative Multithreading (SPSM) architecture is very restrictive in the allowed dependencies between the threads. This usually prevents effective exploitation of thread level parallelism. The Parafrase architecture is very flexible in which program level is partitioned. The benchmark examples showed that procedures and loops are usually the most suitable candidates, while basic blocks mostly contain too many dependencies. However, the Parafrase architecture completely analyses the dependencies at compile-time which restricts the efficiency of the parallelism exploitation.

[2] A solution will be found in one of the first four iterations, because the problem is symmetric.

References

[1] M.S. Lam and R.P. Wilson. Limits of Control Flow on Parallelism. In *Proceedings 19th Annual International Symposium on Computer Architecture*, pages 46–57, 1992.

[2] David W. Wall. Limits of Instruction-Level Parallelism. In *Proceedings of ASPLOS-IV*, pages 176–188, 1991.

[3] K.B. Theobold, G.R. Gao, and L.J. Hendren. On the Limits of Program Parallelism and its Smoothability. In *Proc. 25th Annual International Symposium on Microarchitecture*, pages 10–19, 1992.

[4] D.E. Maydan, J.L. Hennessy, and M.S. Lam. Effectiveness of Data Dependence Analysis. In *Proceedings of the NSF-NCRD Workshop on Advanced Compilation Techniques for Novel Architectures*, 1992.

[5] J. Hordijk and H. Corporaal. A Comparison of Different Multithreading Architectures. Technical Report 1-68340-44(1997)11, Delft University of Technology, Section Computer Architecture and Digital Systems, 1997.

[6] G.S. Sohi, S.E. Breach, and T.N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, 1995.

[7] J. Tsai and P. Yew. The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, pages 35–46, 1996.

[8] P.K. Dubey, K. O'Brien, K. O'Brien, and C. Barton. Single Program Speculative Multithreading (SPSM) Architecture: Compiler-Assisted Fine-Grained Multithreading. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 109–121, 1995.

[9] Milind Girkar and Constantine D. Polychronopoulos. The Hierarchical Task Graph as a Universal Intermediate Representation. *International Journal of Parallel Programming*, pages 519–551, 1994.