

A Microthreaded Architecture and its Compiler

T. Bernard, K. Bousias, B. de Geus, M. Lankamp, L. Zhang, A. Pimentel,
P.M.W. Knijnenburg, and C.R. Jesshope

Computer Systems Architecture Group
Informatics Institute, University of Amsterdam
The Netherlands

{tbernard,bousias,bgeus,mlankamp,zhangli,andy,peterk,jesshope}
@science.uva.nl

Abstract. A different approach to ILP based on code fragmentation, first proposed some 10 years ago, is being used for novel CMP processor designs. The technique, called microthreading, enables binary compatibility across arbitrary schedules. Chip architectures have been proposed that contain many simple pipelines with hardware support for ultra-fast context switching. The concurrency described in the binary code is parametric and a typical microthread is an iteration of a loop. The ISA contains instructions to create a family of micro-threads, i.e., the collection of all loop iterations. In case a microthread encounters a (possibly) long latency operation (e.g., a load that may miss in the cache) this thread is switched out and another thread is switched in under program control. In this way, latencies can effectively be hidden, if there are a sufficient number of threads available. The creation of families of threads is the responsibility of the compiler. In this presentation, we give an overview of the microthreaded model of computation and we show by some small examples that it provides an efficient way of executing loops. Moreover, we show that this model has excellent scaling properties. Finally, we discuss the compiler support required and propose some compiler transformations that can be used to expose large families of threads.

1 Introduction

Recent studies point out that two major problems in scaling up current superscalar microarchitectures are the growing impact of wire delays [1, 5] and the increasing complexity of some critical components such as the issue logic, the bypass network, the register file and the rename logic [19], since they have a direct impact on the clock cycle time. Matzke has argued that in a 80nm process, a signal will take 16 cycles to cover the entire die and at least 20 cycles in a 60nm process [17] which illustrates one of the pitfalls to exponential circuit scaling across time.

A third problem is more subtle but also involves scaling, where a periodic doubling of the chip density scales clock speed as well as transistor concurrency. We are used to microprocessor performance "following Moore's law" but in fact performance improvements have come largely from clock period, which shrinks with linear-dimension scaling, whereas concurrency grows much faster with planar-dimension scaling. Clock speeds however, have also been enhanced by superpipelining over the last decade giving a superscaling of clock speed but this free lunch is all but over. Power density is

directly proportional to frequency and chips are fast approaching thermal limits, which will preclude any further significant increases in clock speed. We are left with exploiting concurrency in instruction execution, a much more difficult challenge, especially if this is to be automated and provide code-compatibility across generations of architectures.

These problems pose a number of significant challenges for future chip architecture. The first and most important is in developing scalable models of concurrency, where the support structures required are scalable in the number of concurrently issued instructions. Secondly, chips must be partitioned into asynchronously-communicating synchronous regions and must tolerate latency in communication, in order to be able to manage the relatively slow signal propagation times. These problems have been solved in grids for large-scale computation but solutions are programmer driven, use low-level tools and not at the granularity required to support instruction-level concurrency. These are indeed big issues, especially when coupled with the requirement for code compatibility.

Chip MultiProcessors (CMPs) have been proposed to solve some of these problems as in a CMP, many processor cores may have their local clock and communicate asynchronously. However, most developments to date have adopted solutions more appropriate for system-level concurrency, rather than instruction-level concurrency. Examples are the Compaq Piranha [2], Stanford Hydra [9] and Hammond et. al. [10], and more recently, commercial designs such as the IBM Power PC [22], Sun Niagara [15] and Intel's Montecito [18].

Other solutions to these challenges have resurrected interest in the data-flow paradigm, as it provides both latency tolerance and asynchronous interfaces. From much research in the 1980s we know that the problems with dataflow include the provision of a sufficiently large and efficient matching store, the inefficiency of managing control structures and the explosion of concurrency that results from executing programs in their least-constrained manner. Finally, this approach requires a functional or single-assignment programming language to capture its semantics. Papadopoulos effectively solved the problem of matching store, by introducing the concept of ETS [20] and recent developments have adopted novel approaches to counter some of the other problems. TRIPS [6], for example, uses dataflow instructions but only within large structured blocks of code, which are scheduled sequentially with conventional control-flow semantics. Because of this partition between dataflow and sequential semantics, the approach does not scale seamlessly. Indeed scaling the hardware will require scaling the hyperblock that provides the dataflow concurrency, which is a compile-time optimization and would require frequent recompilation. Wavescalar [21], on the other hand, adopts a pure dataflow instruction set and attempts to solve the problems of source-language and concurrency explosion. It does this by introducing a wave number across multiple instances of a given context such as a loop or function call. This sequentializes execution and provides a mechanism for resolving multiple writes to the same variable, something not allowed in single assignment languages. In effect it is very similar to the concurrency throttling mechanism that was introduced by Culler and Arvind [7] and which introduces additional dependences to sequentially constrain the execution of k-bounded loops. The Wavescalar approach however, still suffers from inefficien-

cies in managing control flow and will typically execute more instructions for a given computation than are executed in a RISC processor.

This paper concerns another closely related approach based on code fragmentation and called microthreading, where code fragments rather than single instructions are scheduled using dataflow synchronization [13, 11]. The advantage of this approach is that it provides a potential solution to all of the above problems but with increased, rather than decreased efficiency [16]. The increased efficiency is obtained by exploiting regular structures such as loops and their contextual information and offloading their control into hardware rather than coding them inefficiently as dataflow instructions. This provides certain restrictions on the execution model and these together with a brief introduction to the model are discussed in this paper. Background reading on this model can be found at [12].

2 A Microthreaded Architecture

In this section, we give an overview of our concept of *microthreads*. Intuitively, a microthread is one iteration of a loop. The collection of all iterations of a loop is called a *family of microthreads*. A *microthreaded architecture* provides support for executing families of microthreads. It is a distributed, shared-register CMP with many simple in-order pipelines, connected together using an on-chip communications network, see Figure 1.

2.1 Concurrency controls

Instruction	Semantics
Cre	Creates a new family of threads
Swch	Causes a context switch to occur
Kill	Terminates the thread being executed
Bsync	Waits for all other threads to terminate
Brk	Terminated all other threads

Table 1. Concurrency-control instructions

The ISA of a microthreaded architecture is an (existing) RISC ISA together with five new instructions given in Table 1. To discuss the semantics of these new instructions, we give the assembly code for a Livermore hydro fragment, given by the Fortran code

```

DO 1 k = 1, m
1   X(k) = Q + Y(k) * (R * ZX(k+10) + T * ZX(k+11))

```

This fragment would be translated to the assembly in Figure 2. First, an 8 word *Thread Control Block (TCB)* is defined that is used by the `crea` instruction. In particular, the loop structure is defined in this block and the possible values of the loop index

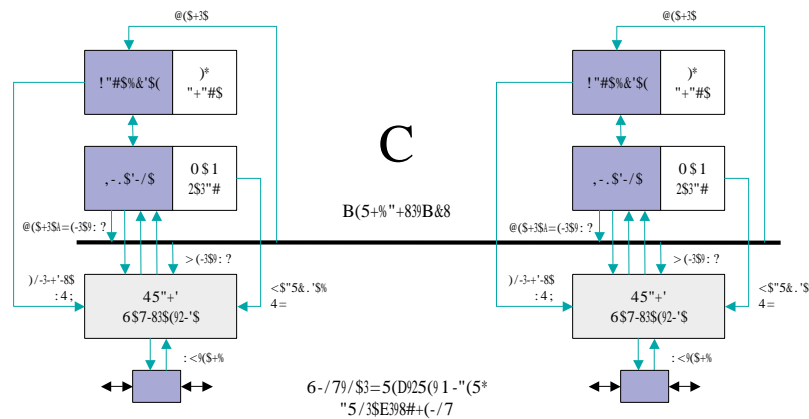


Fig. 1. Overview of microthreaded architecture

are given. Also, a pointer to the code for the loop body is supplied. When the `crea` instruction is executed, schedulers in each processor of a CMP will begin creating a deterministic subset of the iterations on that processor. This involves allocating a set of registers and a slot in the scheduler's *Continuation Queue* to each thread (iteration) as well as initializing the registers to empty and setting the first location, `$L0` to the value of the loop index. Thread creation only happens when there is a sufficient number of resources available and will stop when either all resources have been used or the loop count is exhausted. In the former case new threads will be allocated as resources are released by completed threads.

Each thread is executed in sequential order but different threads can be scheduled to different pipelines. Instructions from different threads scheduled to the same pipeline can be interleaved. Each time a `swch` statement is executed in a thread, that thread is suspended and another thread allocated to the same pipeline can be started or continued. This mechanism allows us to hide latencies: after a load (that may miss in the cache and hence has to be loaded from main memory) we execute a `swch` instruction so that by the time the load is completed, the thread can be resumed again.

It is important to realize that in our model, the memory is decoupled from the pipeline. This means that a load is issued but the pipeline does not stall for it to complete. The load is passed to the memory hierarchy and, on a cache miss, when the load cannot complete in its designated write-back slot, the write-back stage simply sets the target register to *empty* and the data is then written back by the memory system asynchronously. This means that any instruction attempting to read the data when empty

```

        .data                                # create control block
dol:    .word 1                               # start index
        .word m                               # finish
        .word 1                               # stride
        .word 0                               # dependence distance
        .word 3                               # number of global registers
        .word 6                               # number of local registers
        .word 0                               # number of shared registers
        .word body                            # code for threads

main:   lw $G0,Q                             # Main code
        lw $G1,R
        lw $G2,T
        crea dol                             # Create family of threads
        bsync                                # wait till all threads complete
        finish                               # and that's all

body:   lw $L4,ZX+11($L0)                    # Thread code
        lw $L3,ZX+10($L0)                   # load ZX and ZY
        lw $L2,Y($L0)
        mul $L1,$L4,$G2                      # T*ZX(k+11)
        swch                                 # context switch for loads to complete
        mul $L5,$L1,$G1                     # R*ZX(k+10)
        swch                                 # context switch for loads to complete
        add $L1,$L1,$L5                     # Add locally
        mul $L1,$L1,$L2                     # Multiply by Y(k)
        swch
        add $L1,$L1,$G0                     # add to Q
        swch
        sw $L1,X($L0)                       # store result
        kill                                 # thread finishes

```

Fig. 2. Microthreaded assembly code for Livermore hydro fragment

must wait for the data in order to continue. This is detected by reading the register and the `swch` is used only to avoid the inefficiency of flushing out any instructions from the same thread following the attempted read to the empty register. For example, in the code fragment in Figure 2, there is first a load to register `$L4`. Subsequently, this register is used by a `mul` instruction. The compiler is aware at this point that the contents of this register may not be defined because the load may have missed in the cache and so it issues a `swch`. The approach described above adopts the dataflow notion of an *i-structure* to deal with this situation. Essentially, an *i-structure* is a store (in this case, a register) with additional state. Initially, the state of the register is *empty*. When a thread tries to read an empty register, it is suspended but is also ‘attached’ to this register as a *continuation*. The thread is activated again when the register has been filled. When the producer writes the register, it becomes *full* and its value can be read. When the consumer reads a full register, it proceeds normally. Registers are set to empty in the write-back stage of

any instruction that does not produce data with a fixed or statically-schedulable delay, as well as when first allocated.

In the mean time, other threads can be executed. As long as there is a sufficient number of threads available, long latency operations can be hidden efficiently. If the code is single-threaded or when no other thread is scheduled to this pipeline, a `swch` instruction has no effect.

The register file size can be tuned to the latency tolerance required when a processor is designed. Large latencies require a large register file and we have investigated register files up to 1024 entries, which could support some 165 threads of the above code per processor and tolerate some 500 cycles of latency in memory access. We have shown that 5 read/write ports per register file are sufficient to support the pipeline's normal synchronous operation plus all other asynchronous reads and writes to the register file. Moreover, we have shown that a 1024-register file with 5 ports would consume about the same area as the register file for the Alpha 21264 processor [3]. Therefore, such a large register file is not prohibitively expensive. A thread scheduled to one particular pipeline is identified within the pipeline as it executes its instructions and each instruction carries an offset pointer into the register file to access the thread's own set of registers and a local register `$Li` is found relative to this offset.

What characterizes a concurrency model is the communication that it supports. The code described is a vector model of concurrency that executes independent loops concurrently but across multiple processors but, as this loop demonstrates, such independent loops also require communication. In this example we have scalar-vector operations, which are very common in vector code. The scalar values are denoted by the register specifiers `$G0 ... $G2` and it can be seen that these are set in the main thread and read in each of the threads executing a loop instance. Such global communication is normally achieved using a shared register file but this approach does not scale, as it either produces contention on a single port or requires a duplication of ports to match the number of processors. Our solution is to replicate these *global* registers in each processor and whenever a thread writes to one, that value is replicated to each processor by a broadcast bus, as shown in Figure 1. Synchronization on these registers requires multiple continuations to be 'attached' to a register, as multiple threads on a single processor may be constrained to wait for what may be a slow propagation of this global information across the many processors on a chip.

The state for all locally allocated threads is maintained in a memory structure called the Continuation Queue (CQ). This structure maintains a number of mutually exclusive queues, to locate *empty slots*, *active threads*, and *continuations* associated with any register in the local register file. An entry in this queue contains the program counter for this thread together with the addressing offset information.

2.2 Dependences in loops

In many cases, the model defined in the previous section is too weak. It can only deal with independent loops. In this section, we discuss the first extension to the model in order to deal with dependences in loops that have a fixed distance in the iteration space. Consider the following Fortran fragment that executes an inner product sequentially.

```

        .data                # create control block
loop:   .word 1              # start index
        .word n              # finish
        .word 1              # stride
        .word 1              # dependence distance
        .word 0              # number of global registers
        .word 2              # number of local registers
        .word 1              # number of shared registers
        .word body           # code for threads

main:   crea loop
        mv $G0,$S0          # initialize dependence chain
        sw $D0,Q            # store result

body:   lw $L1, A($L0)       # load A(i)
        lw $L2, B($L0)       # load B(i)
        mul $L1,$L1,$L2
        swch
        add $S0,$D0,$L1      # Q' = Q + A(i)*B(i)
        kill

```

Fig. 3. Code fragment with explicit communication between threads

```

DO 1 i = 1,n
1   Q = Q + A(i)*B(i)

```

This fragment would be translated to the code in Figure 3. In the thread control block it is indicated that there exists a dependence with distance 1 between the loop iterations. This means that each iteration is dependent on the previous iteration, using the scalar value Q in the source code. To distribute this algorithm, the code is transformed to make Q a vector quantity, which is distributed to each thread using register $\$S0$. Then, to share data between contexts using a limited register address space, the register $\$S0$ is available as a read-only location in the dependent or consumer context identified by $\$D0$. $\$D0$ in the first iteration is set by the creating context and $\$S0$ from the last iteration can similarly be read from the creating context. In the code for `body`, the last instruction, `add $S0, $D0, $L1`, encapsulates this dependence. It adds the locally generated product to the value of Q , which contains the sum from all previous iteration, read as $\$D0$ and then writes $\$S0$ which passes the sum resulting from this iteration to the next. In this way, the value of Q is passed from thread to thread, constraining the dependence to execute in loop order, even though all threads may execute their loads and multiplication independently and in any order. This captures both ILP across the loop as well as the sequential semantics of the summation.

In order for this to work correctly, a thread has to wait until the previous thread has written its local value of Q , which can occur in one of two ways. If two dependent threads are mapped to a single processor, $\$S0$ and $\$D0$ refer to one and the same register and the full/empty state on that register implements the synchronization. When

dependent threads are mapped to different processors, these two registers are independently allocated, in order to decouple inter-processor communication from pipeline operation. In both cases the reading of the empty $\$D0$ register will suspend the thread on that location. However, if the producer is mapped remotely rather than locally, this will also trigger a hardware mechanism that retrieves and synchronizes with the write to the remote $\$S0$ register in the producer thread.

Distributed algorithms based on constant-strided dependences have been developed to allocate registers and manage thread state for addressing (or mapping) between the registers in different contexts. In this model each thread (if dependent) must have addressing information for its local context and one other thread's context that includes the processor on which to find it. As has already indicated, this information passes down the pipeline with the execution of each thread's instructions and is used to address either a local register file or a remote one using the shared-register network. Since we associate shared and dependent registers by number (i.e., $\$S_i \equiv \D_i), it is known to which remote register a read of a $\$D_i$ register refers.

2.3 Register file partitioning

As indicated above, the address space for each instruction is divided into a number of windows. We have already encountered these windows, namely Global $\$G_i$, Local $\$L_i$, Shared $\$S_i$ and Dependent $\$D_i$. In any implementation of this model a mapping of these windows is identified by the information in the control block and what is generated in the binary code are simply register specifiers constrained by the base ISA's addressing range. However, the information in the control block will determine any special action, such as broadcast or remote read. Each window is described in more detail below and an example mapping is illustrated in Figure 4.

1. Global registers $\$G_i$ can be read and written by all microthreads, including the main or creating thread. Hence these registers can be used to broadcast values and may contain loop independent values.
2. Local registers $\$L_i$ are used only by a single thread.
3. Shared registers $\$S_i$ are used to forward values from one thread to a subsequent thread that depends on it.
4. Dependent registers $\$D_i$ are the registers which are used to receive values from earlier threads.

2.4 Future directions

There are a number of extensions or changes that can be made to the model explained above. Firstly, communication between remote threads is currently based on remote read operations. We could also base data exchange on writes. In this case, the producer thread writes its value into the register of the consumer. A problem the implementation must deal with is the situation when the consumer thread is not yet allocated and any deadlock that this blocking may entail. This can be solved in a number of ways but ultimately this deadlock avoidance may introduce severe inefficiency, as registers will

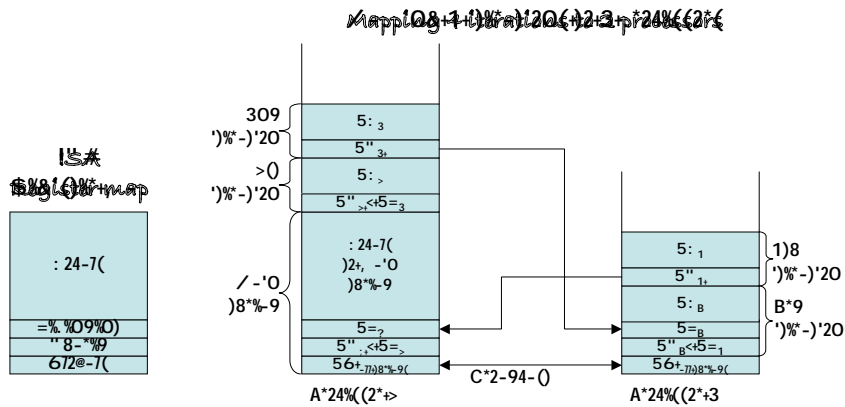


Fig. 4. Register file showing the association between source and destination registers

need to be spilled to allow further allocation. This needs to be investigated in more detail.

Secondly, it is possible to generalize the dependences between threads, so that one thread may communicate with an arbitrary other thread. This would support concurrent gather and scatter operations and potentially make the computation non-deterministic, i.e., when multiple threads write to the same location in the same context. Any suspended thread will only wait for one synchronization and hence read the first value written. This model requires that writes be used for data communication, as we have no mechanism to support multiple remote read continuations to the same location, even though this is quite deterministic. Again this requires further investigation.

We have already defined models that support multiple threads per iteration context, so that several threads may be defined to execute the body of an iteration. In this situation, each thread shares the one context defined for the iteration. Allowing multiple threads to define an iteration, where each thread had its own context, would allow for situations where register allocation to an iteration was constrained by the ISA's register address space. This would probably require arbitrary communication between threads, as both inter-thread communication within an iteration and inter-iteration dependences must be accommodated. Perhaps an intermediate situation would be to allow recursive creation of families of threads, which would support nested loops as well as the situation described above.

3 Compilation Issues

In this section we discuss some of the issues that a compiler faces to generate code for a microthreaded architecture.

3.1 Basic compiler extensions

In our project, we are currently targeting the Suif and MachineSuif compiler suite to our microthreaded processor simulator. This simulator uses Alpha binaries.

This first issue that the compiler needs to do is to recognize independent loops and loops with a constant dependence distance. For these loops, the code generator needs to be extended to generate Thread Control Blocks and the control instructions that are required to fire up the associated families of microthreads. Closely related to this effort is to transform loops to this form, or to move these loops to the innermost position so that they can be executed in a microthreaded manner. This resembles closely the transformations required to generate efficient vector code [4, 23]. However, an alternative way to execute a loop nest is to create a family of microthreads from the outerloop and execute the innerloop(s) as ordinary loops. In this way, the outerloop may carry a dependence with fixed distance. However, a possible drawback is that such a complex microthread may require too many registers so that spill code needs to be inserted. Careful evaluation of the different schemes to deal with loop nests needs to be performed once the simulator and compiler are working.

The second issue to deal with is register allocation. As discussed above, in our current model, the main thread uses 16 general purpose registers instead of 32, as is customary in this ISA. This means that several special registers (like \$31 which holds the return address) need to be remapped. This also implies that the register name space for the main thread and for the microthreads is much smaller than usual. Hence register allocation needs to be carefully tuned in order not to generate too much spill code which could destroy performance. One issue that has to be solved is to determine how many synchronization registers (\$S and \$D registers) are really required by a family of microthreads. Obviously, there exist other ways to deal with the registers. For example, we could assign 32 registers to all threads and access them in the register file relative to a base pointer. A fixed number of registers can be designated in the main thread as registers that hold global values that the spawned microthreads can read. These registers need to be replicated across the processors and when a microthread writes to them, this value needs to be broadcasted. Advantages of this approach are that the changes in the register allocator required are fewer and that special purpose registers can be retained. Another advantage is that all threads have more registers at their disposal so that less spill code needs to be generated. A disadvantage is that many threads do not need the full space of 32 registers so that many registers are wasted. Again, we plan to implement several ways to deal with registers and evaluate these schemes by profiling many applications.

Thirdly, a given loop can be mapped onto several different families of microthreads. For example, the loop in Figure 3 could be mapped onto three families of microthreads in which each of the families executes one of the three load instructions with which the code in Figure 3 begins. One reason we might want to map a loop onto several families

of microthreads is that in this way each family needs less registers so that the restricted register name space doesn't pose to much problems.

3.2 Advanced code transformations

An important issue in the microthreaded model is that there is a sufficient number of microthreads available. The latency of memory references or branches can only be hidden if a context switch can be executed so that a second microthread can execute while the first thread waits for the load to be serviced. This reasoning makes clear that the microthreaded model (at least the basic model we are considering currently) is most suited for loop dominated codes. Many floating point, scientific codes are good candidates for this type of code. Another domain that seems well suited for our purposes are media and DSP codes. These codes have the property that they consist of an alteration of so-called *control code* that set global variables and *loop code* in which most of the execution time is spent. These loops usually are independent loops so that they are highly suited for microthreaded execution.

In order to exploit these loops, we frequently need to rewrite them since in many cases the reference C codes use pointers instead of arrays which are required for most subsequent loop analysis and transformation phases. We plan to implement *array recovery* proposed by Franke and O'Boyle [8] to transform pointer based code to array based code so that the normal compiler analysis and loop level transformations can be applied.

Next, the present model only can deal with single nested loops so that a double nested loop can be converted into a loop in which each iteration a family of microthreads is started. In many cases, this is an inefficient situation. For example, in `mpeg2`, in the motion compensated prediction routines, we have an outerloop that ranges over all macroblocks and for each 8×8 pixel macroblock, a small double loop is executed, like the loop in Figure 5. We would like to execute this loop as a family of microthreads.

```
for (j=0; j<h; j++)
{
  for (i=0; i<w; i++)
    d[i] = (unsigned int)(d[i]+s[i]+1)>>1;
  s+= 1x;
  d+= 1x;
}
```

Fig. 5. mpeg2 code fragment from `predict.c`

However, as it is, we can only execute the innermost `i`-loop which only has 8 iterations. Moreover, it is not parallel as there exist a flow dependence on both `s` and `d`. To deal with this situation, we plan to implement a transformation that is closely related to *loop flattening* [14] that transforms the loop into the form given in Figure 6. The code in Figure 6 is a parallel loop with 64 iterations that can be executed as a family of microthreads. Please note which transformations have been applied.

```

for (k=0; k<h*m; k++)
{
    i = k MOD w;
    j = k DIV w;
    s = s0 + j*lx;
    d = d0 + j*lx;
    d[i] = (unsigned int)(d[i]+s[i]+1)>>1;
}

```

Fig. 6. Modified mpeg2 code fragment from predict.c

1. The double loop is replaced by a single loop (flattening)
2. The original loop indices are recovered by division and modulo operations. These are costly operation but the microthreaded model can hide their latency by context switching. Moreover, in the present case, the loop bounds are 8 so that these operations can be implemented by shifts and masks.
3. The definition of the induction variables `s` and `d` are ‘reverse strength reduced’ so that they can be locally computed for each iteration independently. This operation would of course increase the execution time of the loop were it executed sequentially. In our case, it is an enabling transformation to parallelize the loop.

Next, as mentioned above, there exists an outerloop that ranges over all macroblocks and all operations within one macroblock are independent of the operations in another macroblock. Hence, the operations for all macroblocks can be executed in parallel and if we were able to harvest all this parallelism, we would have a pool of thousands of threads which could be used to hide latencies and speed up the computation for this frame considerably.

In order to do this, we need one more application of loop flattening. However, this flattening cannot be applied immediately, for the following reason. The loop that runs over all macroblocks is located in the top routine `predict`. This routine calls the routine `predict_mb`, which calls the routine `pred`, which calls `pred_comp` which is the routine that contains the fragment from Figure 5. Hence, to exploit the potential parallelism present in the code, we need to aggressively inline three levels of called functions. It should be noted that the resulting loop has a long body containing many `if-then-else` constructs, which may cause the code to run out of the I-cache. Careful tuning of this transformation, possibly pulling `if` statements out of the loop to create several smaller loops that execute `then` or `else` branches, is required.

4 Discussion

In this paper, we have discussed a novel approach to ILP based on code fragmentation, called microthreading. A microthreaded architecture consists of many simple in-order pipelines that can execute these fragments in parallel. We have discussed how one such architecture can be organized and some the issues involved with compilation. Currently we are implementing a base simulator and a compiler for our architecture. Once these

are finished, we will conduct many experiments with many different schemes to deal with the registers, nested loops etc. to select the optimal configuration.

References

1. V. Agarwal, M.S. Hrishikesh, S.W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proc. ISCA*, pages 248–259, 2000.
2. L.A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, Shaz Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *Proc. ISCA*, pages 282–293, 2000.
3. I. Bell, N. Hasaasneh, and C.R. Jesshope. Microgrids and micro-contexts: Support structures for microthread scheduling and synchronisation. In *Proc. 1st Microgrid Conference*, 2005. Submitted to IJPP.
4. A.J.C. Bik. *The Software Vectorization Handbook*. Intel Press, 2004.
5. M.T. Bohr. Interconnect scaling – the real limiter to high performance ULSI. In *Proc. Int'l Electron Devices Meeting*, pages 241–244, 1995.
6. D. Burger, S.W. Keckler, K.S. McKinley, M. Dahlin, L.K. John, C. Lin, C.R. Moore, J. Burril, R.G. McDonald, W. Yoder, and TRIPS Team. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7):45–55, 2004.
7. D. E. Culler and Arvind. Resource requirements of dataflow programs. In *Proc. ISCA*, pages 141–150, 1988.
8. B. Franke and M.F.P. O'Boyle. Array recovery and high-level transformations for DSP applications. *ACM Trans. on Embedded Computing Systems*, 2(2):132–162, 2003.
9. L. Hammond, B.A. Hubbert, M. Siu, M. Prabhu, M. Chen, and K. Olukolun. The stanford hydra cmp. *IEEE Micro*, 20:71–84, 2000.
10. L. Hammond, B.A. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, 30(9):79–85, 1997.
11. C.R. Jesshope. Microthreaded microprocessors: Evolution or revolution? In *Proc. ACSAC*, LNCS 2823, pages 21–45, 2003.
12. C.R. Jesshope. Micro-grids – the exploitation of massive on-chip concurrency. In *Grid Computing, the new frontiers of high-performance computing*, volume 14 of *Advances in Parallel Computing*, 2004.
13. C.R. Jesshope and B. Luo. Micro-threading: A new approach to future RISC. In *Proc. ACSAC*, pages 34–41, 2000.
14. P.M.W. Knijnenburg. Flattening: VLIW code generation for imperfectly nested loops. In *Proc. 7th Workshop on Compilers for Parallel Computers*, pages 254–264, 1998.
15. P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: 32-way multithreaded sparc processor. *IEEE Computer*, 25(2):21–29, 2005.
16. B. Luo and C.R. Jesshope. Performance of a microthreaded pipeline. In *Proc. ACSAC*, 2002.
17. D. Matzke. Will physical scalability sabotage performance gains? *IEEE Computer*, 30(9):37–39, 1997.
18. G. McNairy and R. Bhatia. Montecito: A dual-core, dual-thread itanium processor. *IEEE Computer*, 25(2):10–20, 2005.
19. S. Palacharla, N.P. Jouppi, and J.E. Smith. Complexity-effective superscalar processors. In *Proc. ISCA*, pages 206–218, 1997.
20. G. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor*. Research Monographs in Parallel and Distributed Computing. Pitman/MIT, 1991.
21. S. Swanson, A. Schwerin, A. Petersen, M. Oskin, and S. Eggers. Threads on the cheap: Multithreaded execution in a wavecache processor. In *Proc. WCED*, 2004.

22. J.M. Tandler, J.S. Dodson, J.S. Fields, H. Le, and B. Sinharoy. Power4 system micro-architecture. *IBM Journal of Research and Development*, 46(1):5–25, 2002.
23. H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, 1990.