SMT Issues

- SMT CPU performance gain potential.
- Modifications to Superscalar CPU architecture necessary to support SMT.
- SMT performance evaluation vs. Fine-grain multithreading Superscalar, Chip Multiprocessors.
- Hardware techniques to improve SMT performance:
  - Optimal level one cache configuration for SMT.
  - SMT thread instruction fetch, issue policies.
  - Instruction recycling (reuse) of decoded instructions.
- Software techniques:
  - Compiler optimizations for SMT.
  - Software-directed register deallocation.
  - Operating system behavior and optimization.
- SMT support for fine-grain synchronization.
- SMT as a viable architecture for network processors.
- Current SMT implementation: Intel’s Hyper-Threading (2-way SMT) Microarchitecture and performance in compute-intensive workloads.
Compiler Optimizations for SMT

• Compiler optimizations are usually greatly influenced by specific assumptions about the target micro-architecture.

• Such optimizations devised for single-threaded superscalar/VLIW and shared-memory multiprocessors may have to be applied differently or may not be appropriate when targeting SMT processors due to:
  – The fine-grain sharing of processor resources (functional units, TLB, cache etc.) among threads in SMT processors.
  – Its ability to hide latencies and resulting decrease in idle issue slots.

  – Loop-iteration scheduling and distribution among threads.
  – Software speculative execution.
  – Loop tiling.
### Benchmarks Used

<table>
<thead>
<tr>
<th>Application</th>
<th>Data set</th>
<th>Instructions simulated</th>
<th>L</th>
<th>D</th>
<th>S</th>
<th>E</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>applu</td>
<td>33x33x33 array, 2 iterations</td>
<td>272 M</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hydro2d</td>
<td>2 iterations</td>
<td>474 M</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mgrid</td>
<td>64x64x64 grid, 1 iteration</td>
<td>3.2 B</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>su2cor</td>
<td>16x16x16x16, vector len. 4K, 2 iterations</td>
<td>5.4 B</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>swim</td>
<td>512x512 grid, 10 iterations</td>
<td>419 M</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tomoscav</td>
<td>513x513 array, 5 iterations</td>
<td>189 M</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fft</td>
<td>64K data points</td>
<td>32 M</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LU</td>
<td>512x512 matrix</td>
<td>431 M</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>radix</td>
<td>256K keys, radix 1K, 512K max key value</td>
<td>6 M</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>water-mercuried</td>
<td>512 molecules, 3 timesteps</td>
<td>870 M</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>water-spatial</td>
<td>512 molecules, 3 timesteps</td>
<td>784 M</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>compress</td>
<td>train input set</td>
<td>64 M</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>go</td>
<td>train input set, 2stone9</td>
<td>700 M</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>li</td>
<td>train input set</td>
<td>258 M</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m88ksim</td>
<td>test input set, 4hrystone</td>
<td>164 M</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>perl</td>
<td>train input set, scrabble</td>
<td>56 M</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mmx from NASA7</td>
<td>matrix multiply of 256x128 and 128x64 arrays</td>
<td>29 M</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gmt from NASA7</td>
<td>500x500 Gaussian elimination</td>
<td>354 M</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>adv integration</td>
<td>1Kx1K stencil computation for solving partial differential equations</td>
<td>16 M</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Memory Hierarchy Parameters

<table>
<thead>
<tr>
<th></th>
<th>L1 L-cache</th>
<th>L1 D-cache</th>
<th>L2 cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache size (bytes)</td>
<td>128K / 32K</td>
<td>128K / 32K</td>
<td>16 M / 2 M</td>
</tr>
<tr>
<td>Line size (bytes)</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Banks</td>
<td>8</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Transfer time/bank</td>
<td>1 cycle</td>
<td>1 cycle</td>
<td>4 cycles</td>
</tr>
<tr>
<td>Accesses/cycle</td>
<td>2</td>
<td>4</td>
<td>1/4</td>
</tr>
<tr>
<td>Cache fill time (cycles)</td>
<td>2</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Latency to next level</td>
<td>10 / 18</td>
<td>10 / 18</td>
<td>68</td>
</tr>
</tbody>
</table>

When there is a choice of values, the first (the more aggressive) represents a forecast for an SMT implementation roughly three years in the future and is used in all experiments.

The second set is more typical of today’s memory subsystems and is used to emulate larger data set sizes; it is used in the tiling studies only.
Loop-iteration Distribution

- Concerned with how loop iterations are distributed among different threads in shared-memory multi-threaded or parallel programs.
- Blocked loop iteration distribution or parallelization assigns each thread continuous array data and the corresponding loop iterations.
- Blocked distribution improves performance in shared-memory multiprocessors by increasing data locality thus reducing communication and cache coherence overheads to other processors.
- In SMT processors the entire memory hierarchy is shared reducing the communication costs to share data among the threads active in the CPU core.
- Since threads in SMT share data TLB, blocked assignment has the potential to introduce data TLB trashing due to the possibly large number of TLB needed as the number of threads is increased requiring more data TLB entries.
- Cyclic iteration distribution where iteration and data assignments alternate between threads may reduce the demand on data TLB thus resulting in better performance in SMT processors.
Blocked vs. Cyclic Loop Distribution

a) original loop
   for (j = 1; j < n; j++)
     for (i = 0; i < m; i++)
       u1[j][i] = u2[j][i] + t1/8.0 * Z[j][i]

b) blocked parallelization
   for (j = max(n * myid / numthreads + 1, 1);
       j < min(n * (myid+1) / numthreads + 1, n);
       j++)
     for (i = 0; i < m; i++)
       u1[j][i] = u2[j][i] + t1/8.0 * Z[j][i]

c) cyclic parallelization
   for (j = myid + 1; j < n; j += numthreads)
     for (i = 0; i < m; i++)
       u1[j][i] = u2[j][i] + t1/8.0 * Z[j][i]

d) blocked
   i dimension

Figure 1: A blocked and cyclic loop distribution example.
The code for an example loop nest is shown in a). When using a blocked distribution, the code is structured as in b). The cyclic version is shown in c). On the right, d) and e) illustrate which portions of the array are accessed by each thread for the two policies. (For clarity, we assume 4 threads). Assume that each row of the array is 2KB (512 double precision elements). With blocked distribution (d), each thread accesses a different 8KB page in memory. With cyclic (e), however, the loop is decomposed in a manner that allows all four threads to access a single 8KB page at the same time, thus reducing the TLB footprint.
SMT Performance
Using Blocked Distribution

Figure 2: Speedups over one thread for blocked parallelization.

Table 3: TLB miss rates. Miss rates are shown for a blocked distribution and a 48-entry data TLB. The bold entries correspond to decreased performance (see Figure 2) when the number of threads was increased.
Figure 3: Speedup attained by cyclic over blocked parallelization.
For each application, the execution time for blocked is normalized to 1.0 for all numbers of threads. Thus, each bar compares the speedup for cyclic over blocked with the same number of threads.
Table 4: Improvement (decrease) in TLB miss rates of cyclic distribution over blocked.

<table>
<thead>
<tr>
<th>Application</th>
<th>48-entry TLB</th>
<th>128-entry TLB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of threads</td>
<td>Number of threads</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>applu</td>
<td>0%</td>
<td>50%</td>
</tr>
<tr>
<td>hydro2d</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>mgrid</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>su2cor</td>
<td>14%</td>
<td>99%</td>
</tr>
<tr>
<td>swim</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>tomcatv</td>
<td>0%</td>
<td>-60%</td>
</tr>
</tbody>
</table>
Software Speculative Execution

- Software speculative execution is a compiler code scheduling technique that hides instruction latencies by moving instructions from a predicted branch path to above (before) a conditional branch thus their execution becomes speculative.
  - If the other branch path is taken speculative instructions are cancelled wasting processor resources.
  - Necessary in the case of in-order superscalar and VLIW processors that lack hardware dynamic scheduling.
  - Increased the number of instructions dynamically executed because it relies on speculated instructions filling idle issue slots and functional units.
- SMT processors build on dynamically-scheduled superscalar CPU cores and utilize multiple threads to fill idle issue slots and to hide latencies.
- Software speculation in the case of SMT may not be as useful or may even decrease performance due to the fact that speculated instructions compete with useful non-speculated instructions from the same and other threads.
<table>
<thead>
<tr>
<th>Specfp95</th>
<th>% increase</th>
<th>SPECint95</th>
<th>% increase</th>
<th>SPLASH-2</th>
<th>% increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>applu</td>
<td>2.1%</td>
<td>compress</td>
<td>2.9%</td>
<td>fft</td>
<td>13.7%</td>
</tr>
<tr>
<td>hydro2d</td>
<td>1.9%</td>
<td>go</td>
<td>12.6%</td>
<td>LU</td>
<td>12.5%</td>
</tr>
<tr>
<td>mgrid</td>
<td>0.5%</td>
<td>li</td>
<td>7.3%</td>
<td>radix</td>
<td>0.0%</td>
</tr>
<tr>
<td>su2cor</td>
<td>0.1%</td>
<td>m88ksim</td>
<td>4.0%</td>
<td>water-nsquared</td>
<td>3.0%</td>
</tr>
<tr>
<td>swim</td>
<td>0.1%</td>
<td>perl</td>
<td>4.9%</td>
<td>water-spatial</td>
<td>3.0%</td>
</tr>
<tr>
<td>tomcatv</td>
<td>1.2%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Percentage increase in dynamic instruction count due to profile-driven software speculative execution. Data are shown for 8 threads. (One thread numbers were identical or very close). Applications in bold have high speculative instruction overhead and high IPC without speculation; those in italics have only the former.
<table>
<thead>
<tr>
<th>SPECfp95</th>
<th>spec</th>
<th>no spec</th>
<th>SPECint95</th>
<th>spec</th>
<th>no spec</th>
<th>SPLASH-2</th>
<th>spec</th>
<th>no spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>applu</td>
<td>4.9</td>
<td>4.7</td>
<td>compress</td>
<td>4.1</td>
<td>4.0</td>
<td>fft</td>
<td>6.0</td>
<td>6.4</td>
</tr>
<tr>
<td>hydro2d</td>
<td>5.6</td>
<td>5.4</td>
<td>go</td>
<td>2.4</td>
<td>2.3</td>
<td>LU</td>
<td>6.7</td>
<td>6.8</td>
</tr>
<tr>
<td>mgrid</td>
<td>7.2</td>
<td>7.1</td>
<td>li</td>
<td>4.5</td>
<td>4.6</td>
<td>radix</td>
<td>5.4</td>
<td>5.4</td>
</tr>
<tr>
<td>su2cor</td>
<td>6.1</td>
<td>6.0</td>
<td>m88ksim</td>
<td>4.2</td>
<td>4.1</td>
<td>watersquared</td>
<td>6.4</td>
<td>6.1</td>
</tr>
<tr>
<td>swim</td>
<td>6.5</td>
<td>6.2</td>
<td>perl</td>
<td>3.8</td>
<td>3.7</td>
<td>waterspatial</td>
<td>6.4</td>
<td>6.5</td>
</tr>
<tr>
<td>tomcatv</td>
<td>6.2</td>
<td>5.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 6: Throughput (instructions per cycle) with and without profile-driven software speculation for 8 threads.** Programs in bold have high IPC without speculation, plus high speculation overhead; those in *italics* have only the former.
Figure 4: Speedups of applications executing without software speculation over with speculation

a) SPECfp95

Bars that are greater than 1.0 indicate that no speculation is better.

b) SPLASH2 and SPEC95 int

Bars that are greater than 1.0 indicate that no speculation is better.
Loop Tiling

- Loop tiling (sometimes also referred to as data blocking) aims at improving cache performance by rearranging data access into small tiles or blocks in the most inner loop level that match the size of cache.
  - Reduced average memory access time or latency.
  - Requires additional code and loop nesting levels (tolerated in single-threaded superscalars due to low instruction throughput).
- In SMT loop tiling not as useful as in single-threaded processors because:
  - SMT has better latency-hiding capability.
  - The additional tiling instructions may increase execution time due to SMT’s higher instruction throughput.
- The optimal tiling pattern (blocked or cyclic) and tile size for SMT also may differ than those for single-threaded architectures.
  - Blocked tiling: Each thread is given a private tile thus no data sharing, higher cache size requirements.
  - Cyclic tiling: Threads share tiles by distributing loop iterations over threads, lowers cache size requirements possibly leading to higher performance.
/* Before */

for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        {r = 0;
         for (k = 0; k < N; k = k+1){
             r = r + y[i][k]*z[k][j];};
         x[i][j] = r;
        };

• Two Inner Loops:
  – Read all NxN elements of z[ ]
  – Read N elements of 1 row of y[ ] repeatedly
  – Write N elements of 1 row of x[ ]

• Capacity Misses can be represented as a function of N & Cache Size:
  – 3 NxNx4 => no capacity misses; otherwise ...

• Idea: compute BxB submatrix that fits in cache
Miss Rate Reduction Techniques: Compiler-Based Cache Optimizations

Blocking Example (continued)

/* After */
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i = i+1)
    for (j = jj; j < min(jj+B-1,N); j = j+1)
        {r = 0;
         for (k = kk; k < min(kk+B-1,N); k = k+1) {
            r = r + y[i][k]*z[k][j];
         }
         x[i][j] = x[i][j] + r;
        }

• B is called the *Blocking Factor*
• Capacity Misses from $2N^3 + N^2$ to $2N^3/B + N^2$
• May also affect conflict misses

From 551
a) original loop

```c
for (j = 1; j < N; j++)
    for (k = 1; k < L; k++)
        for (i = 1; i < M; i++)
            c[j][k] += a[i][k]*b[j][i];
```

b) blocked tiling

```c
ub = min(N*(myid+1)/numthreads+1, N);
lb = max(N*myid/numthreads+1, 1);
for (jT=lb; jT <= ub; jT+=jTsize)
    for (kT=1; kT <= L; kT+=kTsize)
        for (iT=1; iT < M; iT+=iTsize)
            for (j=jT; j <= min(N,jT+jTsize-1); j++)
                for (k=max(1, kT);
                    k <= min(L,kT+kTsize-1); k++)
                    for (i=iT; i <= min(M,iT+iTsize-1); i++)
                        c[j][k] += a[i][k]*b[j][i];
```

c) cyclic tiling

```c
ub = min(N * (myid + 1)/numthreads + 1, N);
lb = max(N * myid / numthreads + 1, 1);
for (jT = lb; jT <= ub; jT += jTsize)
    for (kT = 1; kT <= L; kT += kTsize)
        for (iT = 1; iT < M; iT += iTsize)
            for (j = jT+myid; j <= min(N,jT+jTsize-1);
                j += numthreads)
                for (k=max(1, kT);
                    k <= min(L,kT+kTsize-1); k++)
                    for (i=iT; i <= min(M,iT + iTsize-1);
                        i++)
                        c[j][k] += a[i][k]*b[j][i];
```

Figure 6: Code for blocked and cyclic versions of a tiled loop nest.
Figure 7: A comparison of blocked and cyclic tiling techniques for multiple threads. The blocked tiling is shown in a). Each tile is a 4x4 array of elements. The numbers represent the order in which tiles are accessed by each thread. For cyclic tiling, each tile is still a 4x4 array, but now the tile is shared by all threads. In this example, each thread gets one row of the tile, as shown in b). With cyclic tiling, each thread works on a smaller chunk of data at a time, so the tiling overhead is greater. In c), the tile size is increased to 8x8 to reduce the overhead. Within each tile, each thread is responsible for 16 of the elements, as in the original blocked example.
Figure 5: Tiling results with the larger memory subsystem and separate tiles/thread. All the horizontal axes are tile size. A tile size of 0 means no tiling; sizes greater than 0 are one dimension of the tile, measured in array elements. The vertical axes are metrics for evaluating tiling: dynamic instruction count, total execution cycles and AMAT.
Figure 8: Tiling performance of 8-thread mxm. Tile sizes are along the x-axis. Results are shown for a) blocked tiling and the larger memory subsystem, b) blocked tiling with the smaller memory subsystem, and c) cyclic tiling, also with the smaller memory subsystem.
Levels of Parallelism in Program Execution

Level 5: Jobs or programs (Multiprogramming)

Level 4: Subprograms, job steps or related parts of a program

Level 3: Procedures, subroutines, or co-routines

Level 2: Non-recursive loops or unfolded iterations

Level 1: Instructions or statements

Increasing communications demand and mapping/scheduling overhead

Coarse Grain
Medium Grain
Fine Grain

Higher degree of Parallelism

From 756
Computational Parallelism and Grain Size

• Grain size (granularity) is a measure of the amount of computation involved in a task in parallel computation:
  
  – Instruction Level:
    • At instruction or statement level.
    • 20 instructions grain size or less.
    • For scientific applications, parallelism at this level range from 500 to 3000 concurrent statements.
    • Manual parallelism detection is difficult but assisted by parallelizing compilers.
  
  – Loop level:
    • Iterative loop operations.
    • Typically, 500 instructions or less per iteration.
    • Optimized on vector parallel computers.
    • Independent successive loop operations can be vectorized or run in SIMD mode.
Computational Parallelism and Grain Size

- **Procedure level:**
  - Medium-size grain; task, procedure, subroutine levels.
  - Less than 2000 instructions.
  - More difficult detection of parallel than finer-grain levels.
  - Less communication requirements than fine-grain parallelism.
  - Relies heavily on effective operating system support.

- **Subprogram level:**
  - Job and subprogram level.
  - Thousands of instructions per grain.
  - Often scheduled on message-passing multicomputers.

- **Job (program) level, or Multiprogramming:**
  - Independent programs executed on a parallel computer.
  - Grain size in tens of thousands of instructions.
Shared Address Space (SAS) Parallel Programming Model

• Naming: Any process can name any variable in shared space.

• Operations: loads and stores, plus those needed for ordering and thread synchronization.

• Simplest Ordering Model:
  – Within a process/thread: sequential program order.
  – Across threads: some interleaving (as in time-sharing).
  – Additional orders through synchronization.
  – Again, compilers/hardware can violate orders without getting caught.
  – Different, more subtle ordering models also possible.

From 756
Synchronization

Mutual exclusion (locks):
- Ensure certain operations on certain data can be performed by only one process at a time.
- Room that only one person can enter at a time.
- No ordering guarantees.

Event synchronization:
- Ordering of events to preserve dependences
  - e.g. producer $\rightarrow$ consumer of data
- 3 main types:
  - point-to-point
  - global
  - group

From 756
Need for Mutual Exclusion

- Code each process executes:
  - load the value of diff into register r1
  - add the register r2 to register r1
  - store the value of register r1 into diff

- A possible interleaving:

  \[
  \begin{aligned}
  P1 & & P2 \\
  r1 & \leftarrow & \text{diff} & & \{P1 \text{ gets 0 in its } r1\} \\
  r1 & \leftarrow & \text{r1+r2} & & \{P2 \text{ also gets 0}\} \\
  \text{diff} & \leftarrow & \text{r1} & & \{P1 \text{ sets its } r1 \text{ to 1}\} \\
  \text{diff} & \leftarrow & \text{r1} & & \{P2 \text{ sets its } r1 \text{ to 1}\} \\
  \end{aligned}
  \]

- Need the sets of operations to be atomic (mutually exclusive)

From 756
SMT support for fine-grain synchronization

- The performance of a multiprocessor’s synchronization mechanisms determine the granularity of parallelism that can be exploited on that machine. Synchronization on a conventional multiprocessor carries a high cost due to the hardware levels at which synchronization and communication must occur (e.g., main memory). As a result, compilers and programmers must decompose parallel applications in a coarse-grained way in order to reduce synchronization overhead.

- The paper ("Supporting Fine-Grain Synchronization on a Simultaneous Multithreaded Processor", by Dean Tullsen et al. Proceedings of the 5th International Symposium on High Performance Computer Architecture, January 1999, pages 54-58.) proposes and evaluates new efficient fine-grain synchronization for SMT that offers an order of magnitude improvement in the granularity of parallelism made available with this new synchronization, relative to synchronization on conventional shared-memory multiprocessors.
SMT vs. Shared-memory Multiprocessors

- A simultaneous multithreading processor differs from a conventional multiprocessor in several crucial ways that influence the design of SMT synchronization:

1. Threads on an SMT processor compete for all fetch and execution resources each cycle. Synchronization mechanisms (e.g., spin locks) that consume *any* shared resources without making progress, can impede other threads.

2. Data shared by threads is held closer to the processor, i.e., in the thread-shared L1 cache; therefore, communication is dramatically faster between threads. Synchronization must experience a similar increase in performance to avoid becoming a bottleneck.

3. Hardware thread contexts on an SMT processor share functional units. This opens the possibility of communicating synchronization and/or data much more effectively than through memory.
Existing Synchronization Schemes

• Most common synchronization mechanisms are spin locks, such as test-and-set.
  – While test-and-set modifies memory, optimizations typically allow the spinning to take place in the local cache to reduce bus traffic.
• Lock-Free synchronization has been widely studied and is included in modern instruction sets, e.g., the DEC Alpha’s load-locked (ldl l) and store-conditional (stl c) (collectively, LL-SC).
  – Rather than achieve mutual exclusion by preventing multiple threads from entering the critical section, lock-free synchronization prevents more than one thread from successfully writing data and exiting the critical section.
• The Tera and Alewife machines rely on full/empty (F/E) bits associated with each memory block.
  – F/E bits allow memory access and lock acquisition with a single instruction, where the full/empty bit acts as the lock, and the data is returned only if the lock succeeds.
• The M-Machine, attaches full/empty bits to registers:
  – Synchronization among threads on different clusters or even within the same cluster is achieved by a cluster-local thread explicitly setting a register to empty, after which a write to the register by another thread will succeed, setting it to full.
Goals for SMT Synchronization

• Motivated by the special properties of an SMT processor properties, synchronization on an SMT processor should be:

1 **High Performance.** High performance implies both high throughput and low latency. Full/empty bits on main memory provides high throughput but high latency.

2 **Resource-conservative.** Both spin locks and lock-free synchronization consume processor resources while waiting for a lock, either retrying or spinning, waiting to retry. To be resource-conservative on a multithreaded processor, stalled threads must use no processor resources.

3 **Deadlock-free.** We must avoid introducing newforms of deadlock. SMT shares the instruction scheduling unit among all executing threads and could deadlock if a blocked thread fills the instruction queue, preventing the releasing instruction (from another thread) from entering the processor.

4 **Scalable.** The same primitives should be usable to synchronize threads on different processors and threads on the same processor, even if the performance differs. Full/empty bits on registers are not scalable.

• None of the existing synchronization mechanisms meets all of these goals when used in the context of SMT.
A New Mechanism for Blocking SMT Synchronization

- The new proposed synchronization mechanism uses hardware-based blocking locks.
- A thread that fails to acquire a lock blocks and frees all resources it is using except the hardware context itself.
- A thread that releases a lock upon which another thread is blocked causes the blocked thread to be restarted.
- The actual primitives consist of two instructions:
  - Acquire(lock) – This instruction acquires a memory-based lock. The instruction does not complete execution until the lock is successfully acquired; therefore, it appears to software like a test-and-set that never fails.
  - Release(lock) – This instruction writes a zero to memory if no other thread in the processor is waiting for the lock; otherwise, the next waiting thread is unblocked and memory is not altered.
- These primitives are common software interfaces to synchronization (typically implemented with spinning locks).
- For the SMT processor, these primitives are proposed to be implemented directly in hardware.
Hardware Implementation of Proposed Blocking SMT Synchronization Mechanism

- The proposed synchronization instructions are implemented with a small processor structure associated with a single functional unit. The structure, called a lock-box, has one entry per context (per hardware-supported thread).

- Each entry contains: the address of the lock, a pointer to the lock instruction that blocked and a valid bit.

- When a thread fails to acquire a lock (a read-modify-write of memory returns nonzero), the lock address and instruction id are stored in that thread’s lock-box entry, and the thread is flushed from the processor after the lock instruction.

- When another thread releases the lock, hardware performs an associative comparison of the released address against the lock-box entries. On finding the blocked thread, the hardware allows the original lock instruction to complete, allowing the thread to resume, and invalidates the blocked thread’s lock-box entry.

- A release for which no thread is waiting is written to memory.
Hardware Implementation of Proposed Blocking SMT Synchronization Mechanism

- The acquire instruction is restartable. Because it never commits if it does not succeed, a thread that is context-switched out of the processor while blocked for a lock will always be restarted with the program counter pointing to the acquire or earlier.
- Flushing a blocked thread from the instruction queue (and pre-queue pipeline stages) is critical to preventing deadlock.
  - The mechanism needed to flush a thread is the same mechanism used after a branch misprediction on an SMT processor.
- We can prevent starvation of any single thread without adding information to the lock box simply by always granting the lock to the thread id that comes first after the id of the releasing thread.
- The entire mechanism is scalable (i.e., it can be used between processors), as long as a release in one processor is visible to a blocked thread in another.
Evaluating Synchronization Mechanism Efficiency

• The test proposed for evaluating Synchronization mechanism efficiency is a loop containing a mix of loop-carried dependent (serial) computation and independent (parallel) computation, that can represent a wide range of loops or codes with different mixes of serial and parallel work:

<table>
<thead>
<tr>
<th>single-threaded:</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i = 0; i &lt; N; i++)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>parallelized:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(for each thread)</td>
</tr>
<tr>
<td>for (i = threadId; i &lt; N; i += numThreads)</td>
</tr>
<tr>
<td>temp = independent_computation</td>
</tr>
<tr>
<td>acquire(lock[threadId])</td>
</tr>
<tr>
<td>release(lock[nextId])</td>
</tr>
</tbody>
</table>

Figure 1. Our synchronization efficiency test

• The synchronization efficiency metric is the ratio of parallel-to-serial computation at which the threaded version of the loop begins to outperform a single-threaded version.
Synchronization Test configurations

• In Figure 2 *Single-thread* is the performance of the serial version of the loop, which defines the break-even point.
• *SMTblock* is the base SMT synchronization with blocking acquires using the lock-box mechanism.
• *SMT-ll/sc* uses the lock-free synchronization currently supported by the Alpha. To implement the ordered access in the benchmark, the acquire primitive is implemented with load locked and store conditional and the release is a store instruction.
• *SMP-* each use the same primitives as SMT-block, but force the synchronization (and data sharing) to occur at different levels in the memory hierarchy.
Figure 2. The speed of synchronization configurations.
Performance Analysis of synchronization configurations

From the synchronization configurations performance in figure2:

• Synchronization within a processor is more than an order of magnitude more efficient than synchronization in memory.

• The break-even point for parallelization is about 5 computations for SMT-block, and over 80 for memory-based synchronization.

• Thus, an SMT processor will be able to exploit opportunities for parallelism that are an order of magnitude finer than those needed on a traditional multiprocessor, even if the SMT processor is using existing synchronization primitives (e.g., the lock-free LL-SC). However, blocking synchronization does outperform lock-free synchronization;
Performance Analysis of synchronization configurations

- For this benchmark the primary factor is not resource waste due to spinning, but the *latency* of the synchronization operation.
- The observed critical path through successive iterations of the for loop when the independent computation is small and performance is dominated by the loop-carried calculation.
  - In that case the critical path becomes the locked (serial) region of each iteration.
  - For the lock-free synchronization, the critical path is at least 20 cycles per iteration A key component is the branch misprediction penalty when the thread finally acquires the lock and the LLSC code stops looping on lock failure.
  - For blocking SMT synchronization, the critical path through the loop is 15 cycles. This time is dominated by the restart penalty (to get a blocked thread’s instructions back into the CPU).
- In summary, fine-grained synchronization, when performed close to the processor, changes the available granularity of parallelism by an order of magnitude.
Faster Blocking SMT Synchronization Using Speculative Restart

- The restart penalty for a blocked SMT acquire assumes that the blocked thread is not restarted until the corresponding release instruction retires.
- It then takes several cycles to fetch the blocked thread’s instruction stream into the processor.
- While the release cannot perform until it retires (or is at least guaranteed to retire), it is possible to speculatively restart the blocked thread earlier; the thread can begin fetching and even execute instructions that are not dependent on the acquire.
- This optimization further reduces that critical path length of the restart penalty (to get a blocked thread’s instructions back into the CPU).
Figure 3:
The performance of speculative restart

![Graph showing the performance of speculative restart](image)

- SMT-ll/sc
- SMT-block
- SMT-specRel
- single-thread
Figure 4.
The Speedup on Two *Espresso* Loops.
**Espresso Loops Performance Analysis**

- Espresso. For the SPEC benchmark *espresso* and the input file ti.in, a large part of the execution time is spent in two loops in the routine *massive count*.
- The first loop is a doubly-nested loop with both ordered and un-ordered dependences across the outer loop. The first dependence in this loop is a pointer that is incremented until it points to zero (the loop exit condition). The second dependence is a large array of counters which are conditionally incremented based on individual bits of calculated values.
- Figure 4 (Loop 1) shows that both SMT versions perform well; however, there is little performance difference with speculative restart because collisions on the counters are unpredictable (thus restart prediction accuracy is low).
- The second component of *massive count* is a single loop (loop 2) primarily composed of many nested if statements with some independent computation. The loop has four scalar dependences for which we must preserve original program order (shared variables are read and conditionally changed) and two updates to shared structures that need not be ordered.
**Espresso Loops Performance Analysis**

- The *espresso* loop 2 performance of the blocking synchronization was disappointing (Figure 4, loop 2). Further analysis uncovered several contributing factors.
  1. The single-thread loop already has significant ILP.
  2. Most of the shared variables are in registers in the single-thread version, but must be stored in memory for parallelization.
  3. The locks constrained the efficient scheduling of memory accesses in the loop.
  4. The branches are highly correlated from iteration to iteration, allowing our branch predictor to do very well for serial execution; however, much of this correlation was lost when the iterations went to different threads.
- Despite all this, choosing to parallelize this loop still would not hurt performance given our fast synchronization mechanisms.
Figure 5
Speedup for three of the Livermore loops.
Livermore Loops Performance Analysis

- Livermore Loops. Unlike most of the Livermore Loops, loops 6, 13, and 14 are not parallelized by, because they each contain cross-iteration dependencies.
- These loops have a reasonable amount of code that is independent, however, and should be amenable to parallelization, given fine-grained synchronization.
- Loop 6 is a doubly-nested loop that reads a triangular region of a matrix. The inner loop accumulates a sum. While parallelization of this loop (Figure 5, Loop 6), does not make sense on a conventional multiprocessor, it becomes profitable with standard SMT synchronization, and more so with speculative restart support.
- Loop 13 has only one cross-iteration dependence, the incrementing of an indexed array, which happens in an unpredictable, data-dependent order. Loop 13 achieves more than double the throughput of single-thread execution with SMT synchronization and the speculative restart optimization. Here LL-SC also performs well, since the only dependence is a single unordered atomic update.
- Loop 14 is actually three loops, which are trivial to fuse. This maximizes the amount of parallel code available to execute concurrently with the serial portion. The serial portion is two updates to another indexed array, like Loop 13.
Figure 6.
Blocking synchronization performance for Livermore loops with varying number of threads.