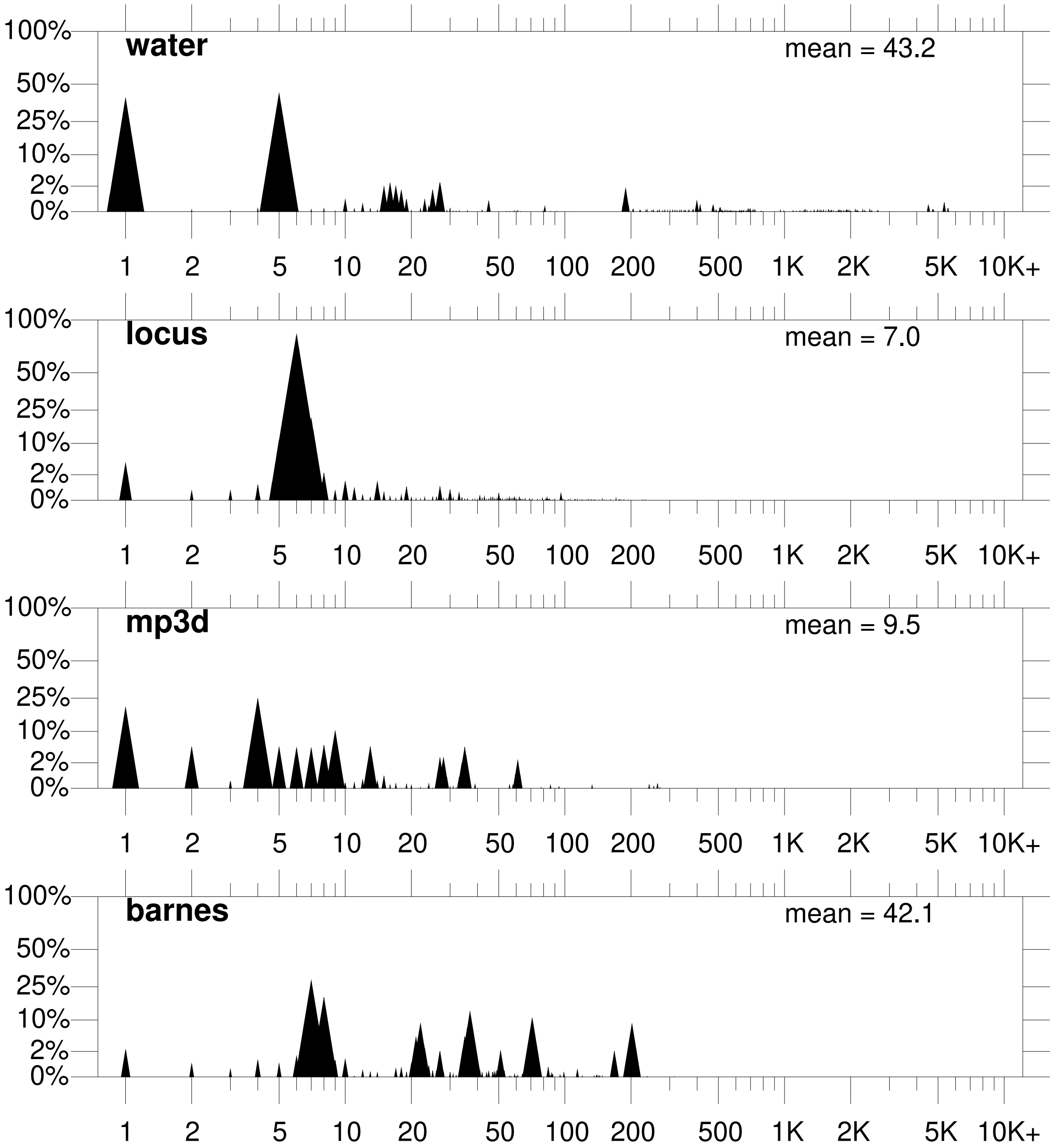



Figure 3.6: Histograms of the run-lengths distributions of the applications running under **switch-on-load**.

| Experiment: **switch-on-load** | | | |
|---|---|---|---|
| Application | Processors | Multithreading | • Latency = 200 cycles<br>• Context switch = 0 cycles<br>• Scheduling = round robin<br>• No shared memory caches |
| sieve | 16 | 1–20 | |
| blkmat | 32 | 1–20 | |
| sor | 8 | 1–50 | |
| ugray | 8 | 1–20 | |
| water | 10 | 1–34 | |
| locus | 2 | 1–40 | |
| mp3d | 8 | 1–40 | |
| barnes | 12 | 1–20 | |

Table 3.2: Experimental parameters for **switch-on-load**.

short mean run-lengths, we can expect `sor`, `locus`, and `mp3d` to require high multithreading levels to keep the processor busy. Also, `ugray` and `water` will require extra multithreading because of the large number of short run-lengths in their distributions.

## 3.3 Testing the Multithreading Model

In this section we compare the performance predicted by the multithreading model to the actual performance observed in simulations. The parameters of the simulation experiments are shown in table 3.2. We have assumed a 200 cycle remote access latency and a context switch cost of 0 cycles[1].

Many of the applications will require a large multithreading level in order to reach high execution efficiencies, but the total number of threads available is limited by the fixed problem sizes that we are able simulate (as discussed in Chapter 2). Therefore, for each application, we have taken the multithreading level ($M$) needed in order to achieve high efficiency, and selected the number of processors so that $P \cdot M$ is approximately equal to the thread limit. The results are presented here as if a just a single set of experiments were performed, but, in fact, preliminary experiments were also performed in order to determine the multithreading levels needed by the applications.

For some applications, such as `locus`, the resultant number of processors used was quite small. In later experiments, with better multithreading models that require fewer threads per processor, we will increase the number of processors used in our simulations.

[1]The zero cycle context switch is justified in Chapter 8.

Figures 3.7 & 3.8 show the predicted and observed performance of the applications under the **switch-on-load** multithreading model. The predicted performance is based on the multithreading model and the run-lengths distributions presented in the previous section. The observed performance is based on the simulations.

The multithreading model predicts *processor utilization* rather than our preferred metric of *execution efficiency*, which we use for most of the results presented in this dissertation. The model predicts processor utilization because the run-length distributions fed into it reflect the entire parallel execution. Some of this execution may include extra operations performed by the parallel program that are not performed by the sequential program. These extra operations keep the processor busy, but they do not contributed to application speedup. Figures 3.7 & 3.8 show both the processor utilizations and the execution efficiencies observed in the simulations. For some applications (`water`, `mp3d`, and `locus`) the processor utilizations and executions efficiencies are indiscernible from each other. For the others, the gap between utilization and efficiency arises because of the extra operations done by the parallel programs. `Locus` and `ugray`, for example, both do dynamic job scheduling and use spinning to wait for jobs to become available. This spinning keeps the processors busy but does not perform useful computation. `Sieve`, `blkmat`, and `sor` also exhibit a gap between processor utilization and efficiency. These applications do static partitioning of the work among the threads. Each thread does the partitioning calculation, and thus with more threads, more time is spent doing these partitioning calculations that are not needed by the sequential programs. All of the applications actually have parallel overheads. They are just much more visible for `sieve`, `blkmat`, and `sor` because these applications have shorter execution times than the other applications.

For most of the applications, there is a also large gap between the processor utilization predicted by the multithreading model and the processor utilization observed in the simulations. This gap arises because the processors sometimes sit idle or underutilized while threads wait on synchronization or because of imperfect load balancing. The jaggedness in the processor utilization curves for `sor` and `water` is an indicator of this load imbalance problem. Another inaccuracy of the multithreading model is that it assumes that the run-lengths drawn from the distribution are mutually independent. In actuality, the applications proceed through different phases of their computations; some phases have short run-lengths, and some phases have long run-lengths. All of these reasons contribute to the optimistic predictions of the multithreading model.

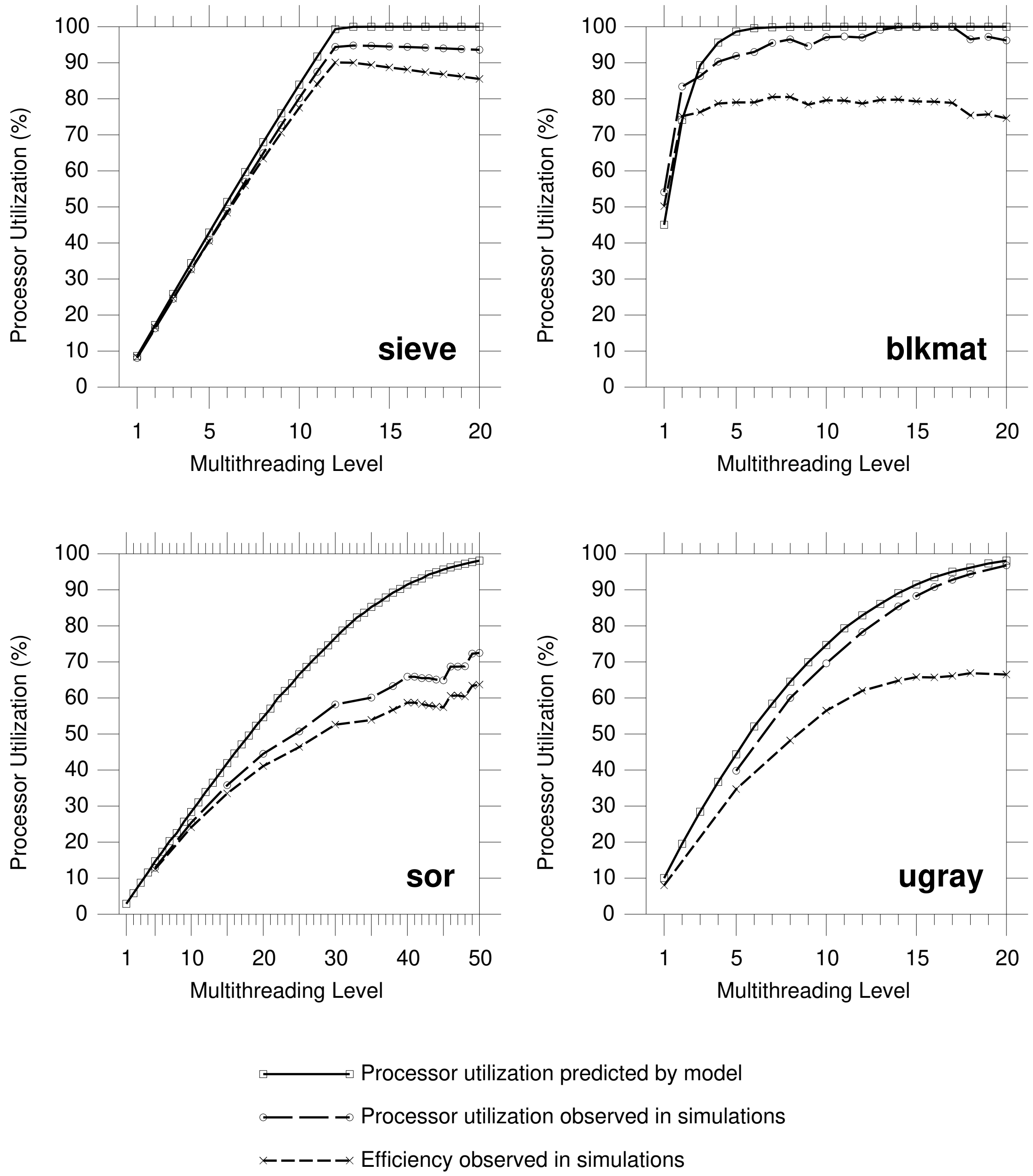



Figure 3.7: Predicted and observed performance for **switch-on-load**.

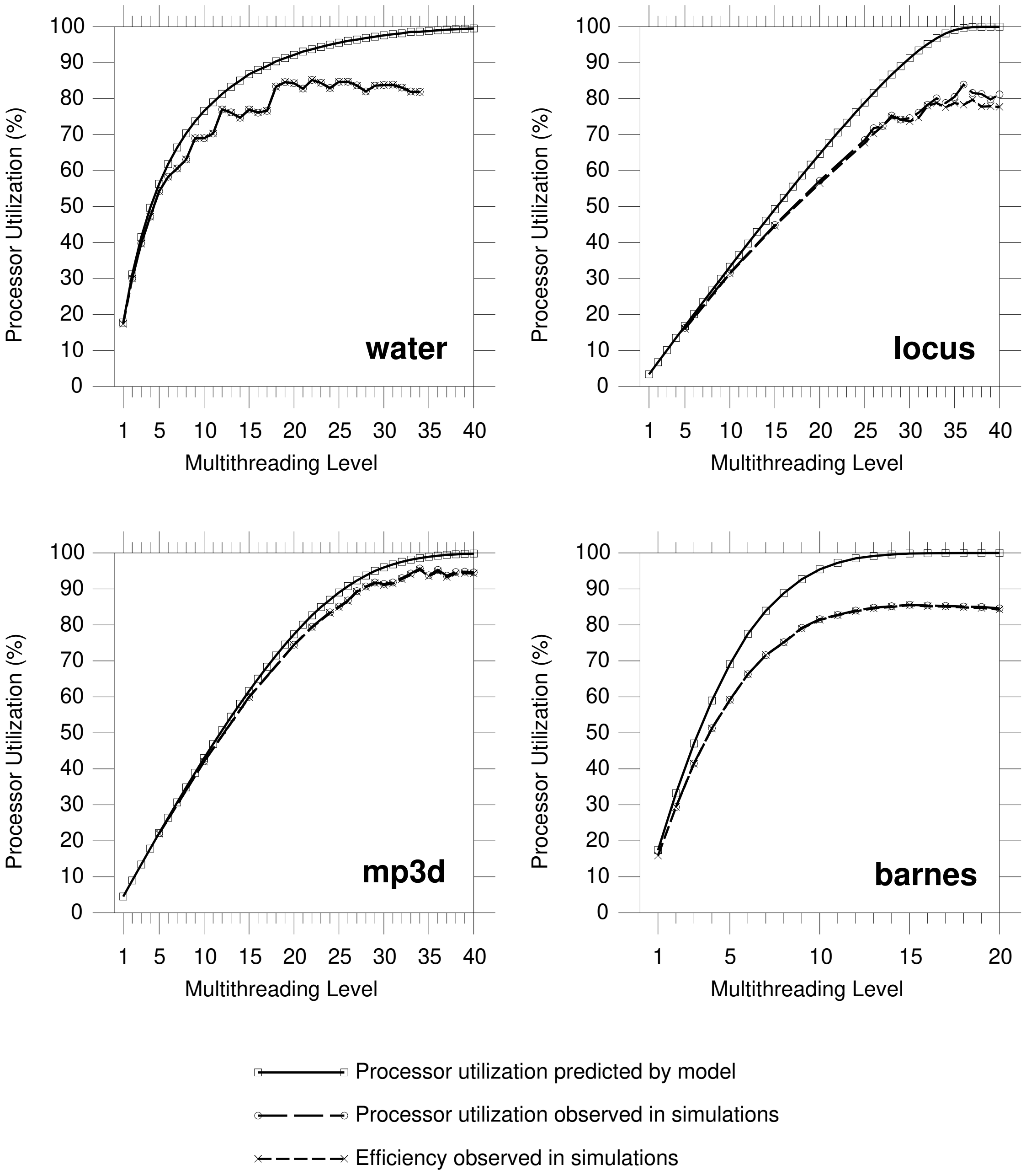



Figure 3.8: Predicted and observed performance for **switch-on-load**.

`Sor` has a very strong correlation between threads. The threads all have a repeating pattern of four short run-lengths followed by a long run-length. Since the threads all start in synchrony, when one thread has a short run-length, so do all the others. Likewise the threads all have their long run-lengths during the same scheduling cycle. If the threads were not so well synchronized, long and short run-lengths would be intermixed and the overall latency hiding would improve. The exact synchrony can be jumbled by random scheduling, and for `sor` it improves the efficiency at $M = 40$ from 59% to 73%. Other applications and other multithreading models do not benefit from this random scheduling, and thus we do not pursue it further.

Usually the performance model predicts processor utilizations higher than what is achieved. An exception occurs for `blkmat` at multithreading levels of 1 or 2. Unlike the rest of the applications, the run-length distribution for `blkmat` varies considerably with the problem size and number of threads. `Blkmat` partitions the computation into blocks (as described in Section 2.2.2), with one block per thread. It makes a local copy of the shared data used within a block, and thus with larger blocks (fewer threads), the average run-length between shared memory references increases. The run-length distribution used by the performance model was obtained from a simulation of 32 processors at a multithreading level of 4, and thus the simulations at smaller multithreading levels had longer run-lengths.

## 3.4 Conclusions

The performance model presented in this chapter provides intuition into the behavior of multithreaded systems. If run-lengths are constant, a multithreading level of $M = 1 + L/R$ is needed to hide the latency. For other distributions, more multithreading is required, particularly for those distributions with short run-lengths.

For real applications, the situation is more complex. Unlike our mathematical model of program behavior, real applications have varying behavior over time, and their threads are not independent. In `sor`, for example, there are alternating phases of computation and convergence checking. The convergence checking phases have little computation and thus shorter average run-lengths than the computation phases. Also, threads synchronize with each other and wait (because of load imbalance) for other threads to complete their calculations. While modeling does give some insights into the behavior to expect from multithreaded systems, in the rest of this dissertation we present only the simulation results.

# Chapter 4

# Multithreading Without Caching

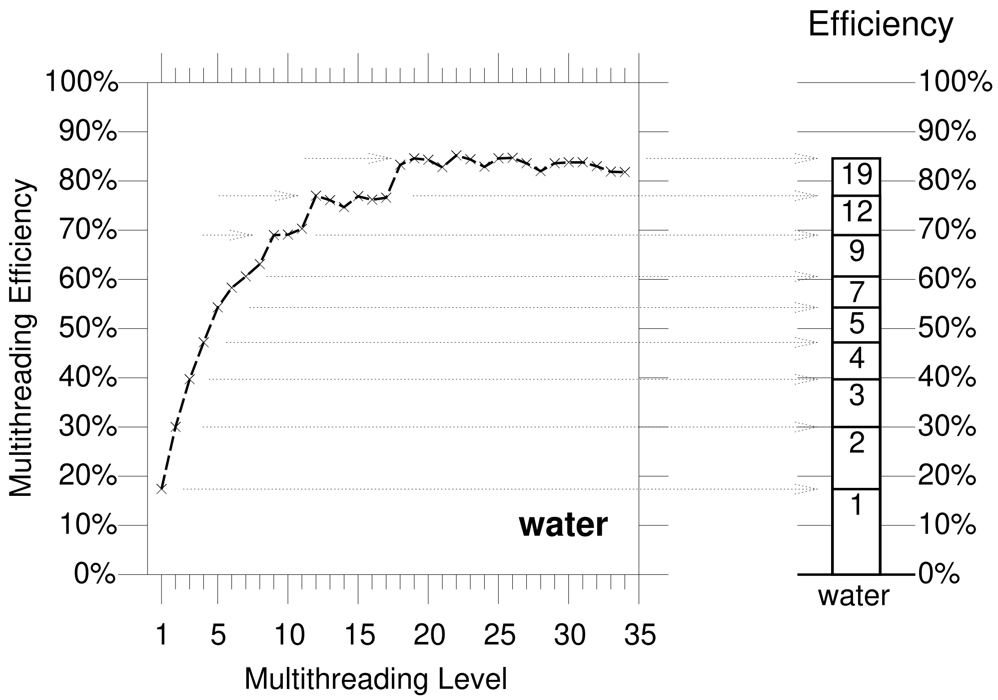
In this chapter we continue the evaluation of **switch-on-miss** and also evaluate the **explicit-switch** multithreading model. These are machine models that do not provide caching of shared data. Their advantage over systems with caching is that they avoid the cost and complexity of cache coherency. Their disadvantages are that they will context switch more frequently, require more threads per processor, and use more network bandwidth.

The **switch-on-load** model will turn out to perform poorly compared to **explicit-switch** because it context switches too frequently. The **explicit-switch** model solves this problem by providing a mechanism for the compiler to group together several shared accesses. The processor can then issue the entire group of messages into the network before it context switches.

## 4.1 New Format

The previous chapter gave performance results for simulations of **switch-on-load**. These results were presented as plots of efficiency versus multithreading level, as shown in Figure 4.1(a). We are mainly interested in the level of performance that can be obtained and the multithreading level needed to obtain it. For future results, we therefore condense this graph into a single bar as shown in Figure 4.1(b). The height of the bar shows the efficiency obtainable, and the number at the top indicates the multithreading level required. The other lines indicate the efficiencies obtainable at lesser multithreading levels.

In this example, the highest bar we show is for $M = 19$. At this level the efficiency




(a) old format (b) new format

Figure 4.1: New format for presenting multithreading efficiency results.

is 84.6%. A slightly higher efficiency of 85.2% is achieved at $M = 22$, but we do not show it because such a minor increase in performance would probably not be worth increasing the multithreading level. Although we may hope that applications will have abundant threads, for many applications threads will be a limited resource. In this and other results presented in this bar graph format, we report the highest efficiency up to the point where the efficiency increases by less than 1% per additional thread.

## 4.2 Switch-On-Load

Figure 4.2 shows the **switch-on-load** multithreading efficiencies in the new format. Many of the applications need large multithreading levels. Particularly high are `sor`($M = 40$), `locus`($M = 32$), and `mp3d`($M = 29$). Furthermore, even at high multithreading levels, some applications are achieving only moderate efficiencies: `sor`(59%), `ugray`(66%).

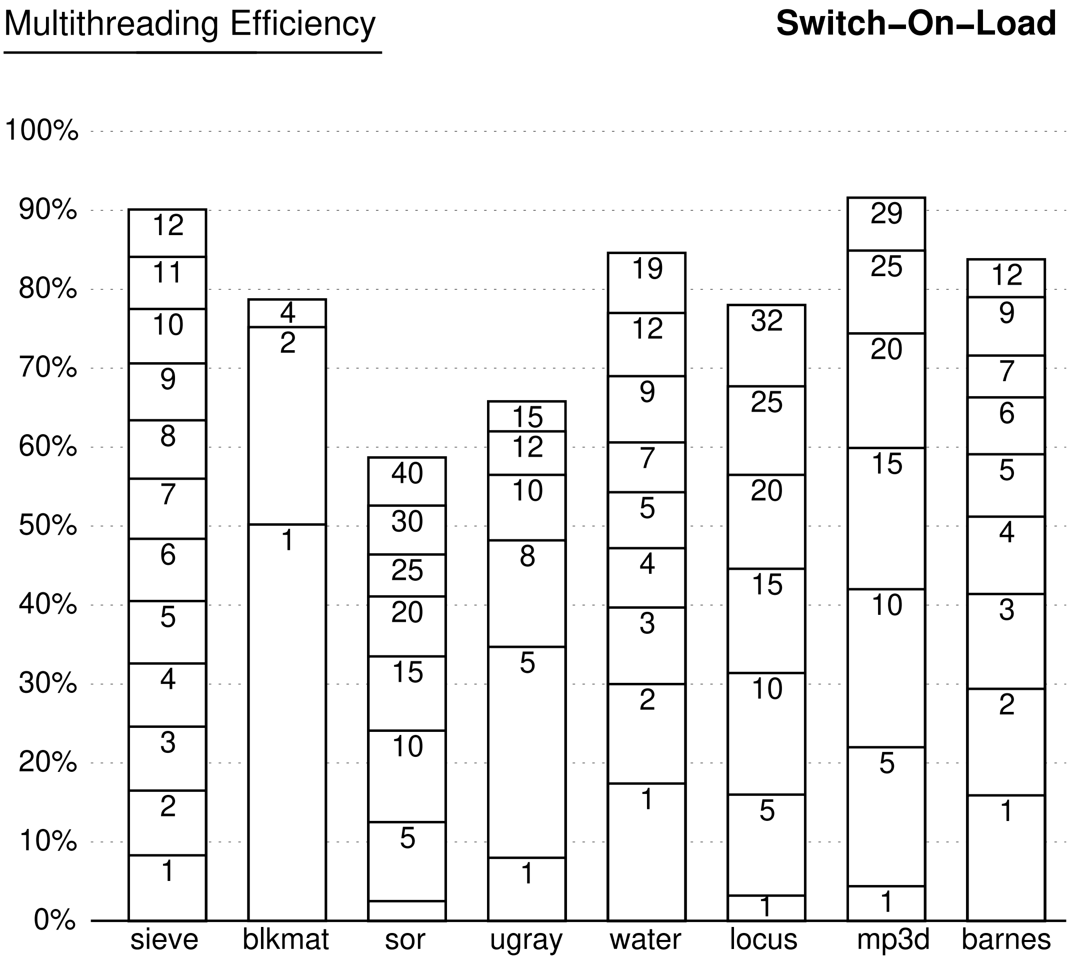



Figure 4.2: Multithreading levels and the efficiencies they achieve under **switch-on-load**.

These high multithreading levels were predicted by the multithreading model because of the short average run-lengths and the many short run-lengths in the distributions. With such high multithreading levels, large problem sizes will be required in order to provide sufficient parallelism. Also the hardware cost to support these multithreading levels will be large because of the large number of register sets required. Machines, such as the HEP[Kow85], have been built that provide more than enough register sets, however decreasing the required multithreading level has many benefits: smaller problem instances can utilize the full set of processors, less hardware is needed to hold the register sets, and less application overhead is incurred when fewer threads are used.

## 4.3 Increasing the Run-Lengths: Explicit-Switch

The key to decreasing the multithreading level and increasing preformance is to increase the run-lengths. This involves both raising the average run-lengths and eliminating short run-lengths. To do this, a thread must be allowed to issue more than one reference into the network before being context switched. There are two multithreading models that address this: **switch-on-use** and **explicit-switch**.

Under **switch-on-use**, the hardware issues remote loads into the network and continues executing the same thread. It context switches only when the thread tries to use a value that has not yet returned. If the compiler can arrange instructions so that several remote loads are grouped together, the loads will all be issued into the network before the thread tries to use any of the results and is forced to context switch. This will eliminate excess context switching and thereby increase run-lengths.

Another way to allow issuing multiple loads before context switching is to provide an explicit context switch instruction that the compiler can insert between the group of loads and the later uses of the requested data. The effect is the same as under **switch-on-use**. The difference is that the hardware may be a little simpler under **explicit-switch** than under **switch-on-load** because it does not have to check the status of registers as they are used. In this section we will explore performance of the **explicit-switch** model. We expect that the results for **switch-on-use** would be virtually identical. Relevant hardware issues are discussed in Chapter 8.

### 4.3.1 Grouping Within Basic Blocks

The inner loop of the `sor` application is shown in Figure 4.3(a) as an example. Without grouping, the 5 loads are issued one at a time, with a context switch after each one. In Figure 4.3(b) the code has been reorganized so that all 5 loads are grouped together and are then followed by a single context switch instruction. Rather than having four short run-lengths followed by one long run-length, there is now just a single long run-length.

A compiler designed for a multithreaded architecture will group shared loads whenever possible. Since the compilers we have today do not do this grouping, we wrote a post-processor which finds the basic blocks in an object file[1], does dependency analysis within the

[1] A basic block is a sequence of instructions that are executed without any branches into or out of it except at the ends.

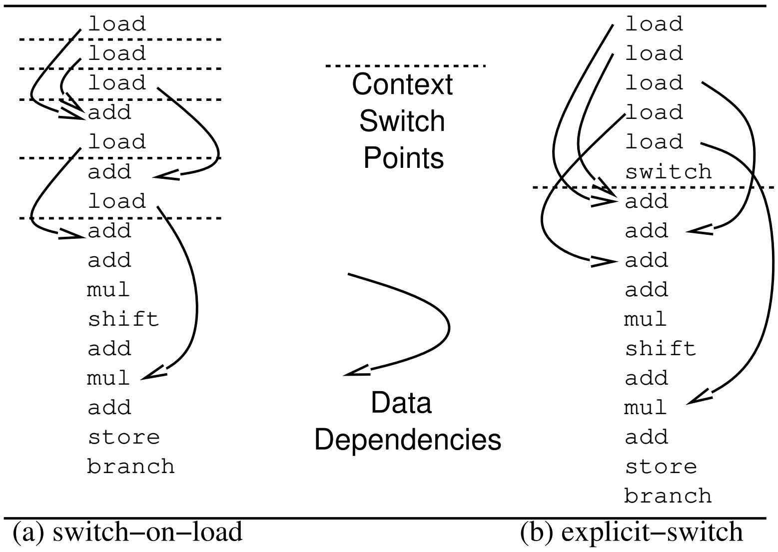



Figure 4.3: Inner loop of `sor` under **switch-on-load** and reorganized for **explicit-switch**.

basic blocks, and then reorganizes the instructions so as to group the shared loads together. The reorganization is performed by percolating shared load instructions upward through the basic block as far as possible while still obeying all data dependencies. Some register renaming is used to eliminate artificial dependencies caused by the original compiler's register assignment, but because we do this analysis at the assembly language level, we must make pessimistic assumptions[2] which restrict our ability to reorganize the code. After the shared load instructions have been percolated upward, the code reorganizer inserts context switch instructions as needed to separate the groups of independent loads from the use of their results. Despite the limited knowledge available when optimizing assembly language, our code reorganization appears to work very well for basic blocks.

Table 4.1 shows the grouping obtained by reorganizing the basic blocks of our benchmark applications. The grouping factor is the average number of shared loads per context switch. For `sor` and `water` this grouping was very successful, grouping on average almost 5 loads between context switches. Grouping in the other applications was less successful, with `locus`, `sieve`, and `blkmat` having only marginal or no grouping at all.

[2]We assume that every shared store might have a conflict with every shared load because of address aliasing.

| Application | Grouping Factor | Mean Run-Length |
|---|---|---|
| sieve | 1.00 | 19.3 |
| blkmat | 1.00 | 164.9 |
| sor | 4.65 | 29.2 |
| ugray | 1.29 | 26.4 |
| water | 4.76 | 206.6 |
| locus | 1.05 | 8.3 |
| mp3d | 2.28 | 22.6 |
| barnes | 1.68 | 71.5 |

Table 4.1: Grouping and mean run-lengths achieved for the applications after reorganization of their basic blocks.

**Experiment: run-lengths for explicit-switch**

| Application | Processors | Multithreading |
|---|---|---|
| sieve | 16 | 12 |
| blkmat | 32 | 4 |
| sor | 16 | 8 |
| ugray | 8 | 12 |
| water | 20 | 6 |
| locus | 2 | 28 |
| mp3d | 16 | 14 |
| barnes | 16 | 8 |

- Latency = 200 cycles
- Context switch = 1 cycle
- Scheduling = round robin
- No shared memory caches

Table 4.2: Experimental parameters for measuring run-lengths for **explicit-switch**.

The new run-length distributions are shown in Figures 4.4 & 4.5; they were obtained from experiments as specified in Table 4.2. These run-length distributions for **explicit-switch** should be compared to the run-length distributions for **switch-on-load** that were shown on pages 40 & 41.

For `sor` and `water` virtually all of the short run-lengths have been eliminated. `Mp3d` and `barnes` still have some short run-lengths, but they show higher mean run-lengths and fewer short run-lengths.

The other four applications show little change. `Locus` had a small amount of grouping that eliminated the shortest (1 or 2 cycle) run-lengths, but this is not very significant because these short run-lengths comprised only 4.5% of the total. The change in

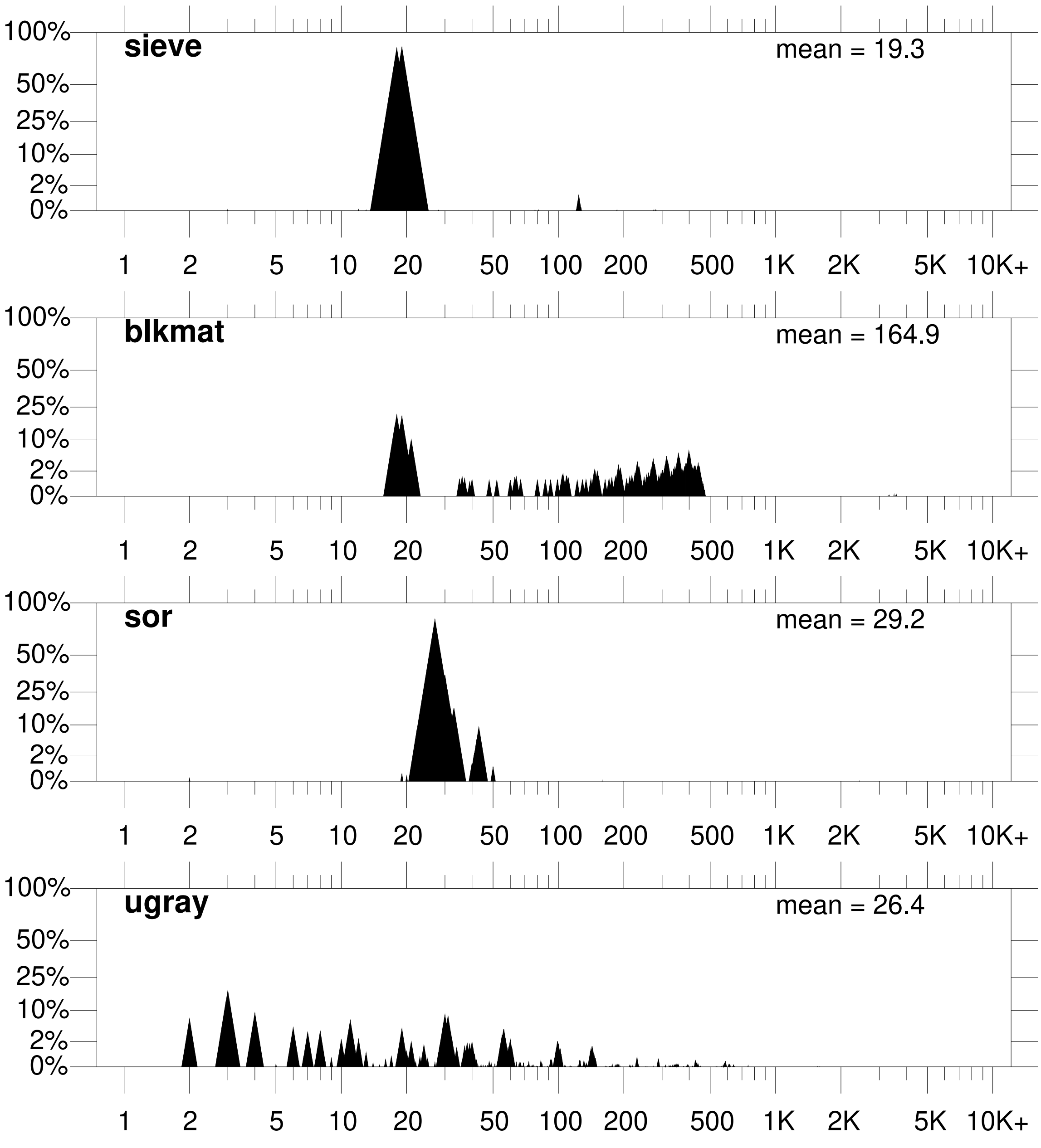



Figure 4.4: Histograms of the run-lengths distributions of the applications running under **explicit-switch**.

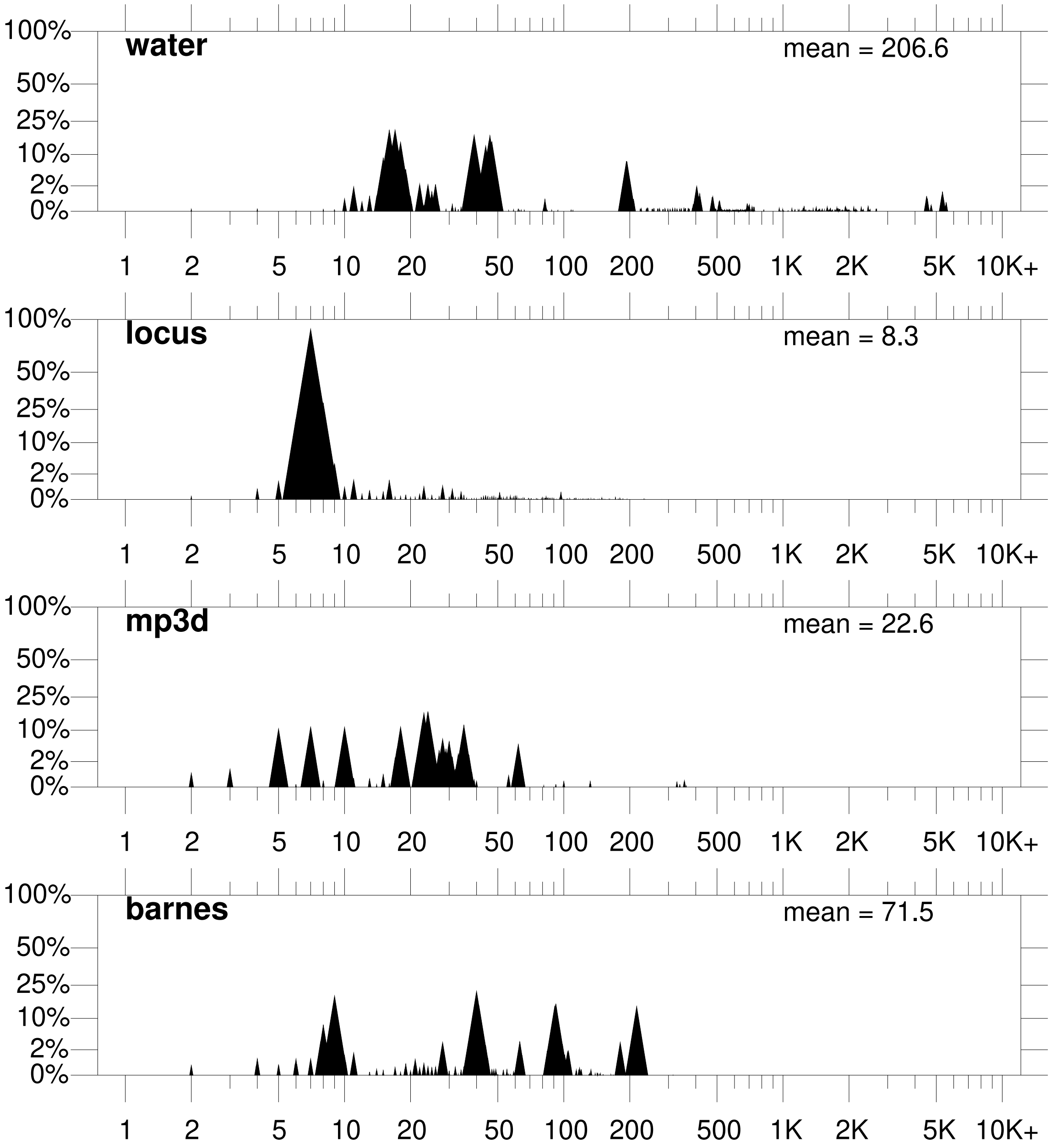



Figure 4.5: Histograms of the run-lengths distributions of the applications running under **explicit-switch**.

| **Experiment: explicit-switch** | | | |
|---|---|---|---|
| Application | Processors | Multithreading | • Latency = 200 cycles<br>• Context switch = 1 cycle<br>• Scheduling = round robin<br>• No shared memory caches |
| sieve | 16 | 1–12 | |
| blkmat | 32 | 1–4 | |
| sor | 16 | 1–8 | |
| ugray | 8 | 1–12 | |
| water | 20 | 1–6 | |
| locus | 2 | 1–28 | |
| mp3d | 16 | 1–14 | |
| barnes | 16 | 1–8 | |

Table 4.3: Experimental parameters for **explicit-switch**.

mean run-length from 7.0 under **switch-on-load** to 8.3 under **explicit-switch** can mainly be attributed to the extra cycle in each run-length from the added switch instruction. This extra cycle is overhead and diminishes performance. The next most troubling application is `ugray`. The grouping factor was only 1.29 and there are still many short run-lengths of just 2, 3, or 4 cycles. These short run-lengths will hamper the efforts of multithreading. The lack of grouping for `sieve` and `blkmat` is unimportant since these applications already had well behaved run-length distributions and moderate or long mean run-lengths.

The experiments used to measure **explicit-switch** execution efficiencies are listed in Table 4.3. There is now a context switch cost of 1 cycle because of the added context switch instructions. We have also increased the number of processors used for `sor`, `water`, `mp3d`, and `barnes`. Under **explicit-switch** they use lower multithreading levels than they did under **switch-on-load**, and thus the surplus threads were used to increase the number of processors. It might seem odd to compare results from **switch-on-load** and **explicit-switch** that use different numbers of processors, nevertheless it is reasonable because there is very little difference in the results when using either the old or new processor numbers. This might seem more obvious if we recall that the multithreading behavior depends on the run-length distributions. For the **switch-on-load** and **explicit-switch** multithreading models the run-lengths usually do not depend on the number of threads used. As long as the number of threads is kept within the limits set by the available parallelism, the number of processors used does not have much impact on the efficiency results obtained. We have chosen to increase the number of processors because it makes the simulations more similar to the way that applications will be run on real machines (with many processors).

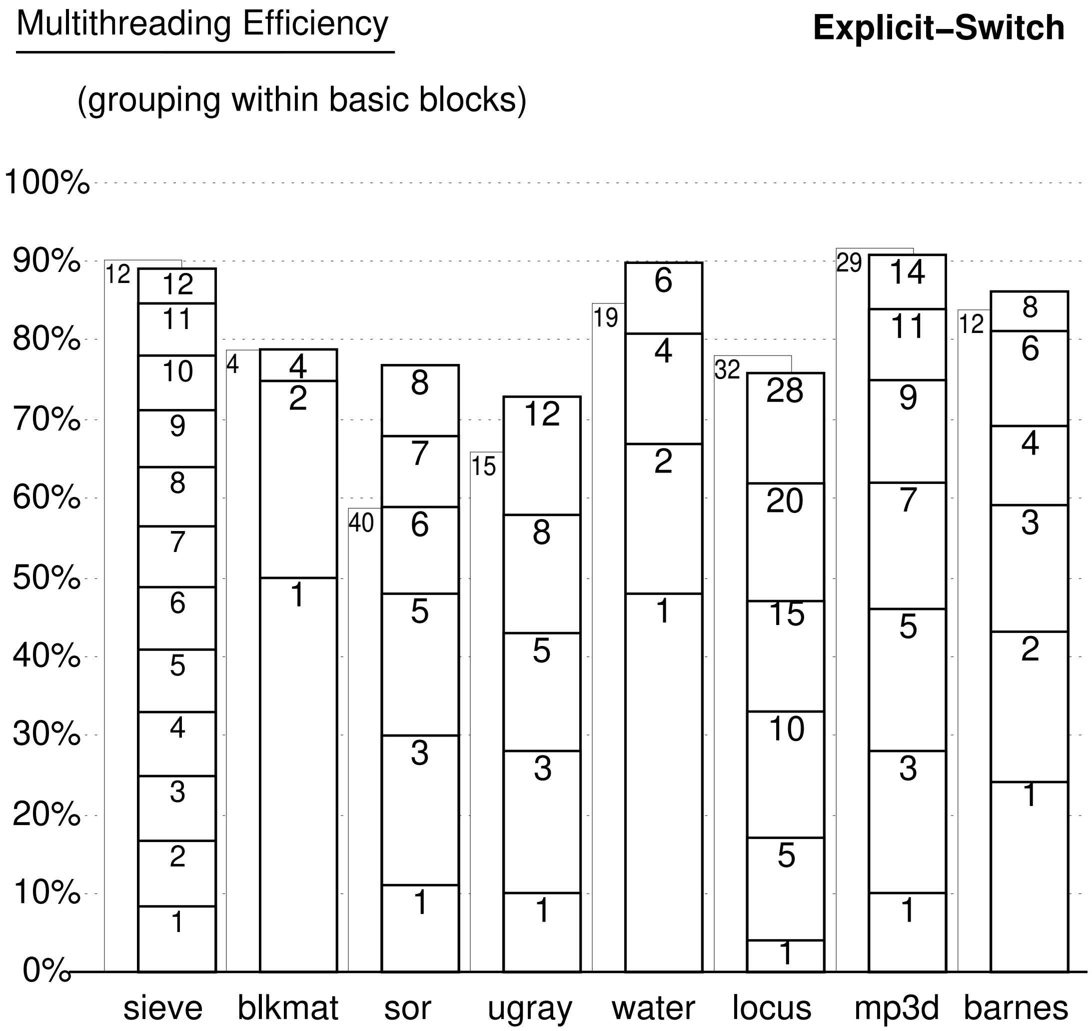



Figure 4.6: Multithreading levels and the efficiencies they achieve under **explicit-switch**. The bars in the foreground show the results for **explicit-switch**, while the bars in the background show the results for **switch-on-load** for comparison.

Figure 4.6 shows the performance results for **explicit-switch**. For each application, two bars are shown. The bar in front shows the efficiency achieved under **explicit-switch** at various multithreading levels. The bar in back shows the previous results from the **switch-on-load** model for comparison. The applications with large grouping factors (`sor`, `water`, and `mp3d`) achieve equal or better performance while using less than half of the multithreading that they needed under **switch-on-load**. `Barnes` and `ugray` had smaller grouping factors, but they were still enough to provide significant decreases in the multithreading levels required. The one disappointment was with `locus`, which had negligible grouping. Its efficiency actually declined because of the overhead of the added switch instructions.

Besides the overhead of 1 cycle for each added switch instruction, there are also software and hardware overheads caused by the grouping of loads. The software overhead arises because the compiler has many optimizations that it must perform. On the MIPS processor, on which these results are based, most floating point operations take more than 1 cycle to execute. These extra cycles are called *delay slots*, and they can be hidden by other instructions, such as loads, if they are available. However by adding the extra constraint of grouping shared memory loads, the compiler has less flexibility in filling these delay slots.

The hardware overhead arises from the increased burstiness of the traffic. Grouped loads are issued into the network in quick succession. The network may not be able to accept a reference every cycle, and thus later references in the group may be delayed. For our simulations, we assume the there is sufficient buffering that the processor never has to stall upon issuing a reference. While overheads do exit, their costs are small compared to the benefit of grouping.

In the `water` application, there are some run-lengths of around 5,000 cycles. It might make sense to try some sort of preemptive scheduling mechanism rather than round-robin scheduling in order to better utilize those long run-lengths for hiding the latency of other threads. This will be explored in Chapter 5.

Overall, the results for **explicit-switch** look very promising. Most of the applications can achieve around 80% efficiency with a multithreading level of 10 or less threads per processor. For a large parallel machine, 80% efficiency is commendable, and a multithreading level of 10 is reasonable for on-chip implementation, as will be shown in Chapter 8. `Locus` and `ugray` are the two applications with inadequate performance. In the next section we will argue that better compiler optimization techniques could improve the grouping

```
t1 = shared_x;
if (t1 > xmax)
  xmax = t1;
t2 = shared_y;
if (t2 > ymax)
  ymax = t2;
```

(a) Code Motion

```
if (shared_flag)
  sum += shared_x;
```

(b) Speculative Loading

```
while (i > 0)
 {
  sum += shared_x[i];
  i--;
 }
```

(c) Loop Unrolling

Figure 4.7: Example code fragments with potential for inter-block grouping.

for these applications and raise their performance to be comparable with the rest of the applications.

### 4.3.2 Grouping Beyond Basic Blocks

In the previous section, our code reorganization and grouping of shared loads was done only within basic blocks. Compiler based optimization could do better by looking beyond the scope of a single basic block.

Figure 4.7 shows three simplified examples of situations taken from the `ugray` and `locus` applications that would be amenable to inter-block grouping by a good optimizing compiler. In these examples, shared variable are prefixed with "`shared_`", and all other variables are local. In example (a), the loading of `shared_y` can be moved upward past the conditional test and grouped with the loading of `shared_x`. In example (b), the loading of `shared_x` could be moved ahead of the `if` statement and grouped with the loading of `shared_flag`. This is called a speculative load since it is done on the speculation that the conditional test will be true and that the load will in fact be needed. In example (c), several iterations of the loop could be unrolled and the exposed multiple loads from the `shared_x` array could then be grouped.

Code motion and loop unrolling are standard optimizations for a good optimizing compiler. Speculative loading, however, is trickier. It might be the case that the conditional test checks the boundary conditions of an array. If the load is moved before the boundary check, it might access off the end of the array and cause an unwarranted memory trap. Rogers and Li[RL92] have proposed a simple mechanism of dealing with this problem by adding a poison bit to each register and taking a trap only upon the use of a poisoned register. A further problem arises if speculative loads are used indiscriminately. If many of

<table>
<tr><th colspan="4">Experiment: explicit-switch with inter-block grouping</th></tr>
<tr><td>Application</td><td>Processors</td><td>Multithreading</td><td rowspan="9"><ul><li>Latency = 200 cycles</li><li>Context switch = 1 cycle</li><li>Scheduling = round robin</li><li>Each thread has a one line cache with a line size of 32 words.</li></ul></td></tr>
<tr><td>sieve</td><td>32</td><td>1–5</td></tr>
<tr><td>blkmat</td><td>64</td><td>1–3</td></tr>
<tr><td>sor</td><td>16</td><td>1–9</td></tr>
<tr><td>ugray</td><td>12</td><td>1–10</td></tr>
<tr><td>water</td><td>20</td><td>1–5</td></tr>
<tr><td>locus</td><td>8</td><td>1–7</td></tr>
<tr><td>mp3d</td><td>32</td><td>1–11</td></tr>
<tr><td>barnes</td><td>16</td><td>1–5</td></tr>
</table>

Table 4.4: Experimental parameters for **explicit-switch** with inter-block grouping.

the loaded values are not actually used, the amount of memory traffic will increase. Because of this, speculative loading will require careful evaluation.

### 4.3.3 Estimation Experiment

Since we do not have a compiler that does inter-block grouping, we have designed a simple experiment to estimate the amount of potential grouping available. The rational for this experiment comes from our inspections of `locus` and `ugray`, the applications from which we are most interested in obtaining additional grouping. We noticed that in these two applications many of the inter-block grouping opportunities came either from small stride accesses to an array or from accesses to multiple fields in a structure. Both of these access patterns will have memory addresses that are close together, and therefore if we detect these access patterns, we can predict which references might be groupable.

A cache acts as a detector of spatially and temporally nearby references, and we have used a very small cache as our detection mechanism. We simulated a one line cache associated with each thread, and any hits in the cache were presumed to have been groupable with the preceding reference. The cache line was 32 words, which is long enough to hold many of the structures in `ugray`. We used a one line cache so as to be as conservative as possible in our estimations. We also added instrumentation so that we could verify the results. For `locus` we were in fact conservative, but for `ugray` it is unclear since there were both erroneous groupings and missed opportunities.

The experimental parameters are summarized in Table 4.4. For `ugray` 42% of the

| Application | Grouping Factor | Mean Run-Length |
|---|---|---|
| sieve | 12.31 | 236.8 |
| blkmat | 1.40 | 208.0 |
| sor | 4.99 | 31.5 |
| ugray | 1.94 | 40.4 |
| water | 7.17 | 311.2 |
| locus | 7.03 | 56.1 |
| mp3d | 3.29 | 33.3 |
| barnes | 3.02 | 129.2 |

Table 4.5: Grouping and mean run-length estimates if the compiler could do inter-block grouping.

loads hit in this cache, and the grouping factor increased from 1.29 to 1.94. For `locus` the hit rate was 84%, and the grouping factor increased from 1.05 to 7.03. This dramatically shows the potential for compiler based grouping. Table 4.5 shows the grouping estimates for all of the applications, and Figure 4.8 shows the expected multithreading levels based on these higher grouping factors.

This experiment is only a means of estimating the available grouping opportunities. To verify the results, we examined the code of `ugray` and `locus` to see where the cache hits were coming from. In `ugray` we found that 47% of the identified grouping opportunities were valid. These were cases where one field in a structure was examined, and if it met some condition, a second field was loaded and used in a computation. A smart compiler could group these two loads together by speculatively loading the second value in the expectation that the test would succeed and thus the second load would be needed. The other 53% of the grouping opportunities identified by the cache experiment turned out to be cases of coincidental memory allocation such as when two small structures were allocated in the same cache line. While examing the code, however, we also found many interblock grouping opportunities that were missed by the cache. Overall, it remains unclear whether this experiment overestimated or underestimated the interblock grouping potential for `ugray`.

For the `locus` application, a single instruction was found to be responsible for 95% of the cache hits. This turned out to be in a loop that was stepping horizontally though a large two dimensional array. A compiler could easily group these loads by unrolling the loop. In addition, we found similar loops that stepped vertically through the array. The

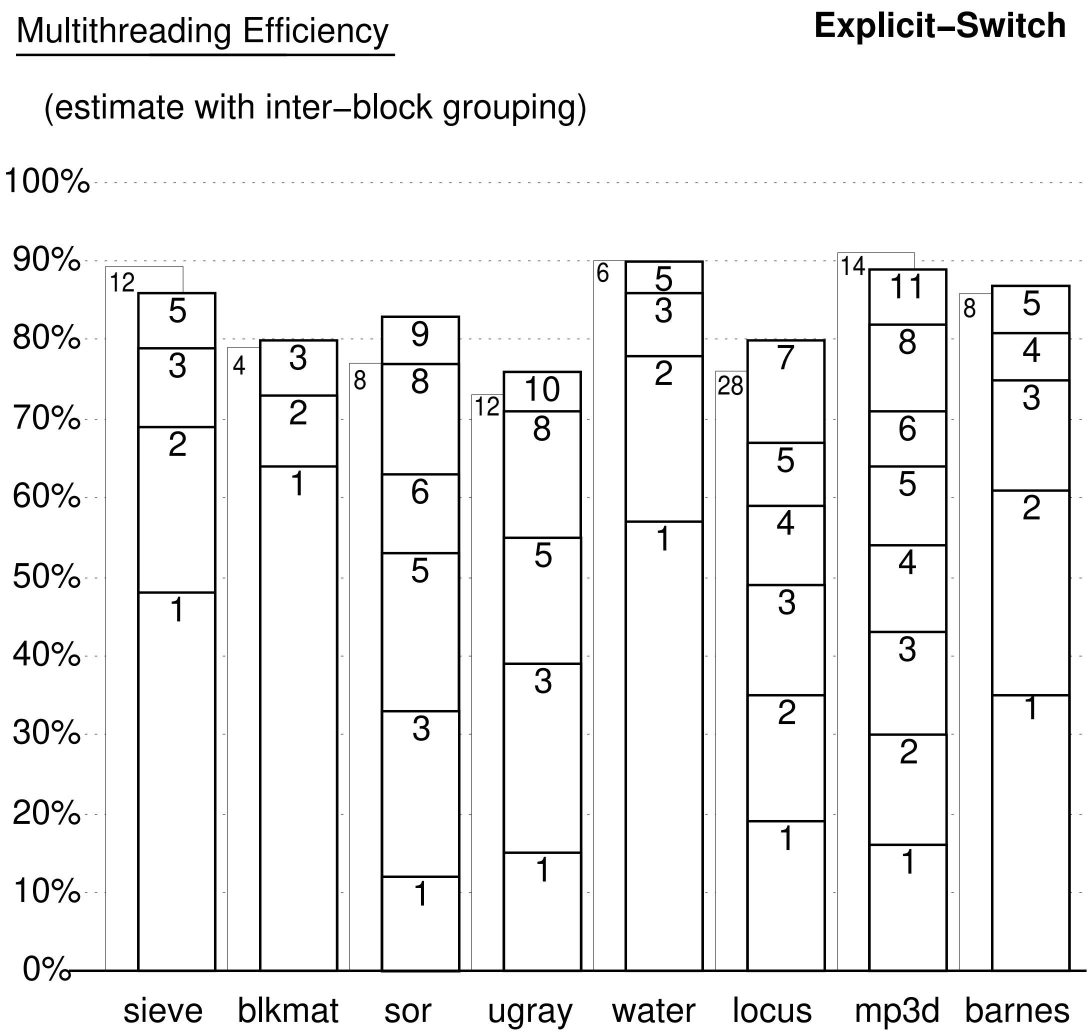



Figure 4.8: Multithreading levels and the efficiencies they achieve under **explicit-switch** with estimated inter-block grouping. The bars in the foreground show the results for **explicit-switch** with inter-block grouping, while the bars in the background show the earlier results for **explicit-switch** without it.

same compiler unrolling technique could group these loads as well, but they were missed by the cache. Thus for `locus` our experiment underestimated the potential for interblock grouping.

For the toy applications (`sieve`, `blkmat`, and `sor`), we have also verified that inter-block grouping is possible. In `sieve` this would involve inter-procedural analysis. In `blkmat` it involves a complex code motion. And in `sor` it involves a simple loop unrolling.

Figure 4.8 shows that with the addition of inter-block grouping, all of the applications can now obtain efficiencies near or above 80% using 10 threads or less per processor. In particular, notice the dramatic improvement of `locus` because of the grouping made possible by loop unrolling.

## 4.4 Conclusions

In this chapter we have shown that multithreading is effective at hiding long latencies to shared memory. The **switch-on-load** model performs poorly for applications that access memory frequently, but the **explicit-switch** model solves this problem by allowing the grouping of independent loads and thereby eliminates many extraneous context switches. For most of our applications grouping within basic blocks is adequate, and for the others there do exist inter-block grouping opportunities. Further research in compiler optimization is needed to fully explore the grouping of accesses.

Simulation results indicate that a multithreading level of 10 threads per processor is adequate for hiding a 200 cycle remote reference latency, and that we can expect efficiencies of 80% or better from a multithreaded parallel machine. This machine provides no hardware caching of shared data, and thus it does not have the complexity of providing cache coherency. The one drawback, which is the subject Chapter 6, is that *all* accesses to shared data are sent across the interconnection network, and thus the network bandwidth requirements will be high.

# Chapter 5

# Multithreading With Caching

Caches are advantageous because they can reduce the number of remote memory references. However, they complicate a parallel machine because they require mechanisms for maintaining coherency among all the caches in the system[ASHH88, CF78, HLRW92, ON90]. In this chapter we first evaluate a system that provides caching but not multithreading, and then evaluate the additional performance improvement possible with **switch-on-miss** and **conditional-switch** multithreading.

## 5.1 Caching

Simulation studies in the literature have reported varying effectiveness for caching on large scale parallel computers. Early DASH results[GHG+91] reported miss rates of 20%, 23%, and 34% for three applications (`mp3d`, `pthor`, and `lu`). While O'Krafka[ON90] reported miss rates of 1.3%, 4.8%, and 2.7% on a different set of applications (`ssim`, `verf`, and `genie`). The higher miss rates in the DASH studies were a result of their choosing three benchmarks all with poor reference locality. Later DASH results[LLJ+92] with a larger set of benchmarks exhibit much lower miss rates (although they are not explicitly reported).

The experimental setup shown in Table 5.1 was used to measure the cache miss rates for our applications. For these simulations, the cache size is 64K bytes, it is 4 way set associative, and it has a 16 byte line size[1]. Coherency is maintained with the Censier and Feautrier[CF78] invalidation based cache coherency protocol. This protocol maintains a full directory of owners for each data line. It serves as a convenient starting point for

[1]The 16 byte line size is shown in Section 7.2 to be the best choice for minimizing bandwidth needs.

**Experiment: caching**

| Application | Processors | Multithreading |
|---|---|---|
| sieve | 128 | 1 |
| blkmat | 64 | 1 |
| sor | 16 | 1 |
| ugray | 32 | 1 |
| water | 29 | 1 |
| locus | 10 | 1 |
| mp3d | 32 | 1 |
| barnes | 32 | 1 |

- Latency = 200 cycles
- Each processor has a 64K byte cache with a 16 byte line size and 4 way set associativity.

Table 5.1: Experimental parameters for caching without multithreading.

comparison with other research, however this may not be the most cost effective choice because of its large hardware cost[ON90].

For simplicity, we continue to assume a latency of 200 cycles for all network references. In reality, references causing coherency traffic would take longer than other references because of the additional message(s) sent to maintain coherency. For example, a straightforward implementation of invalidations would take two round-trip message times (four messages): the request message from the processor to the memory, the invalidation message from the memory to the invalidation site, the acknowledgment message back to the memory, and finally the response message back to the processor. However, a smarter implementation, such as the DASH protocol[LLG$^+$90], can reduce this from four message times to three. Furthermore, in their prototype implementation[LLJ$^+$92] they found the extra latency of a reference requiring coherency to be only 30% over that for a normal reference. Our constant latency assumption is thus slightly optimistic.

Table 5.2 shows the simulation results. For most of the applications the miss rates are just a few percent and caching performs well. The two exceptions are `mp3d` and `blkmat`.

`Mp3d` has *low* reuse of data[2], and its high miss rate is a result of this. It also has a *high* access rate, and thus despite the presence of caches, it still sends a large number of accesses into the network. Without multithreading, it achieves an execution efficiency of only 15%. Gupta *et. al.*[GHG$^+$91] obtained a processor utilization of 26% for this application on their simulations of the DASH multiprocessor[3]. They assumed a latency of less than half

[2] See Section 2.2.7.

[3] This value was calculated based on their results under release consistency, which is similar to our assumption of weak consistency.

| Application | Miss Rate | Efficiency |
|---|---|---|
| sieve | 0.3% | 89% |
| blkmat | 42.5% | 59% |
| sor | 0.9% | 67% |
| ugray | 3.9% | 63% |
| water | 2.9% | 73% |
| locus | 1.8% | 65% |
| mp3d | 16.0% | 15% |
| barnes | 2.3% | 78% |

Table 5.2: Average miss rates and execution efficiencies on a machine with 64K byte caches, 200 cycle latency, but no multithreading.

of what we did, and thus our lower efficiency is to be expected.

`Blkmat` also has a high miss rate (42.5%), but because it has a *low* access rate, the resultant access rate is low enough to allow it to achieve 59% efficiency. `Blkmat` has a low access rate because it was programmed to make local copies of shared data. These local copies can be thought of as software caching, and thus the hardware cache is superfluous.

For the other applications, the efficiencies are in the 60% to 70% range. These efficiencies are acceptable for large parallel machines. For instance, executing at 70% efficiency on a 1000 processor machines would give a speedup of 700. We thus conclude that multithreading is not essential when caching is provided.

Gupta *et. al.*[GHG$^+$91] obtained quite different results in their studies of cache coherent multiprocessors. They looked at just three applications: `mp3d`, `pthor`, and `lu`. These all have high miss rates and low execution efficiencies, as `mp3d` does in our studies. In our larger application suite, `mp3d` is the exceptional case.

Most of our applications achieve acceptable execution efficiencies, but there is still significant performance loss due to latency. Thus there is an opportunity for multithreading to help push execution efficiencies higher. In the subsequent sections we look at the performance improvements that can be obtained by using multithreading to hide the network latency.

<table>
<tr><th colspan="4">Experiment: run-lengths under switch-on-miss</th></tr>
<tr><th>Application</th><th>Processors</th><th>Multithreading</th><td rowspan="9">• Latency = 200 cycles<br>• Context switch = 3 cycles if caused by a miss, 0 cycles if caused by the scheduling policy<br>• Scheduling = lock-priority + spin-switch[4]<br>• Each processor has a 64K byte cache with a 16 byte line size and 4 way set associativity.</td></tr>
<tr><td>sieve</td><td>128</td><td>1</td></tr>
<tr><td>blkmat</td><td>64</td><td>3</td></tr>
<tr><td>sor</td><td>16</td><td>4</td></tr>
<tr><td>ugray</td><td>32</td><td>3</td></tr>
<tr><td>water</td><td>29</td><td>3</td></tr>
<tr><td>locus</td><td>10</td><td>2</td></tr>
<tr><td>mp3d</td><td>32</td><td>11</td></tr>
<tr><td>barnes</td><td>32</td><td>2</td></tr>
</table>

Table 5.3: Experimental parameters for measuring run-lengths under **switch-on-miss**.

## 5.2 Run-Lengths with Caching

Multithreading behaves very different when there is a cache in the system compared to when there is not. Without a cache, under **explicit-switch**, context switches occur at rates ranging from once every 30 cycles, to once every 300 cycle. However with caches, we can now expect most of the previous context switches to be avoided and the mean run-lengths between context switches to rise considerably. Rather than multithreading many threads in order to hide each other's latency, we will need perhaps only two threads per processor so that one can execute while the other is waiting on memory.

The experiments described in Table 5.3 were used to measure the run-length distributions of the applications. The multiple threads on a processor all share the cache, and thus they may interfere with each others' cached data. The miss rates will thus be higher under multithreaded execution than the miss rates listed in Table 5.2 for execution without multithreading[5]. The differing miss rates imply differing run-lengths, and thus the run-length distributions with caching will vary based on the level of multithreading and the size of the caches. We gathered the run-lengths at the multithreading levels that were found to be appropriate for each application.

Figures 5.1 & 5.2 show the run-length distributions under **switch-on-miss**. The

[4]The scheduling policy will be discussed in Section 5.2.1. This policy is non-optimal, but it was chosen because it causes minimum interference with the run-lengths.

[5]Section 7.3 studies the increase in miss rates from multithreading.

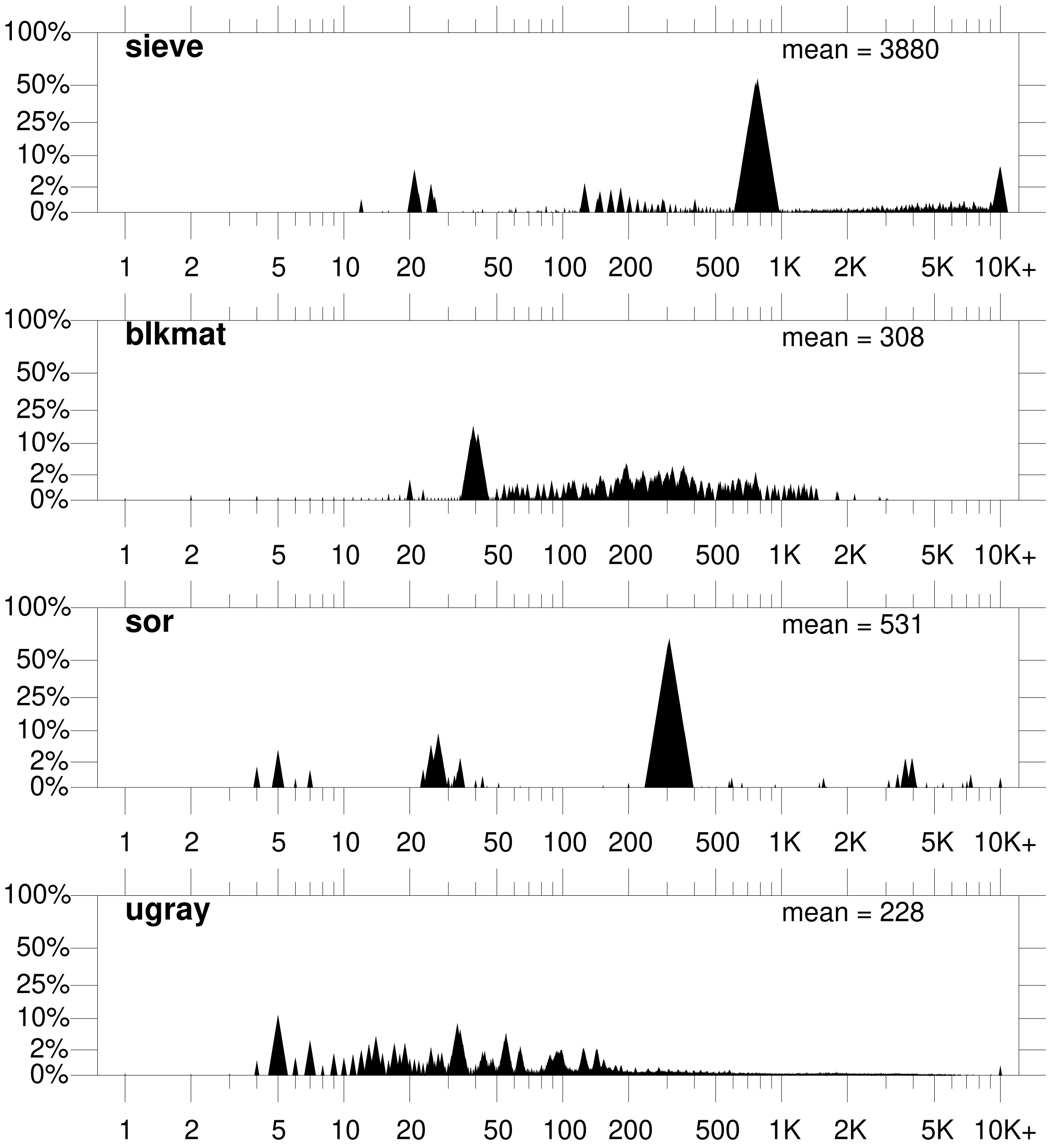



Figure 5.1: Histograms of the run-lengths distributions of the applications running under **switch-on-miss**.

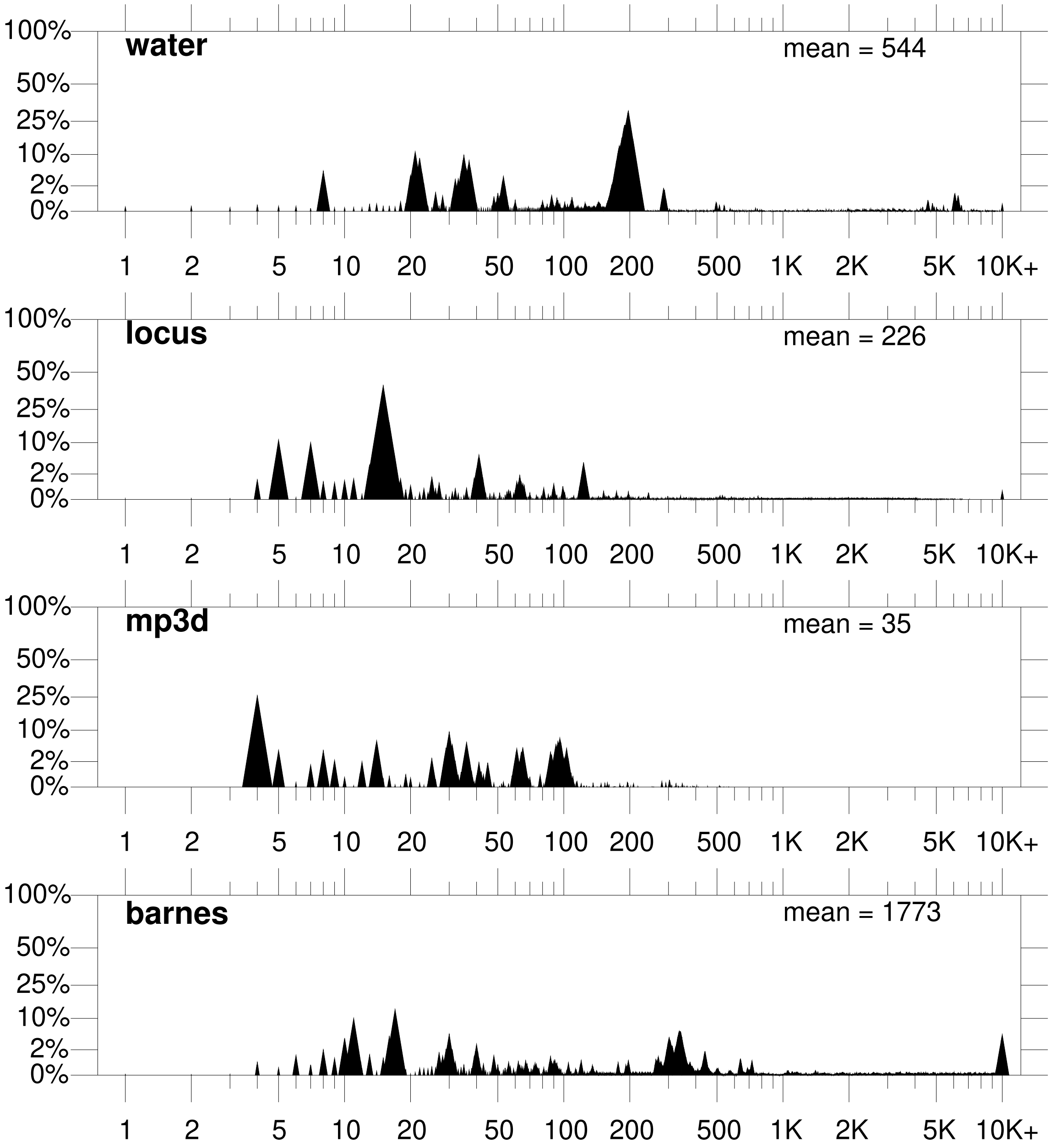



Figure 5.2: Histograms of the run-lengths distributions of the applications running under **switch-on-miss**.

mean run-lengths are now above 200 cycles (except for `mp3d`). If the run-length distributions were constant, M = 2 would be sufficient to completely hide the 200 cycle latency, but unfortunately this is not the case. There are still many short run-lengths where misses occur on successive references. The net effect of the cache is that it raises the average run-lengths and spreads out the run-length distributions. When long sequences of accesses hit in the cache, there are run-lengths that last for thousands or even tens of thousands of cycles.

### 5.2.1 Smarter Scheduling

The disparity in run-lengths suggests that a simple round-robin scheduling policy may no longer be the best choice. Long run-lengths can cause problems because they block out other threads from the processor. Consider the following scenarios with two threads on a processor:

**unbalanced scenario:** Thread $A$ is executing with long run-lengths taking thousands of cycles, while thread $B$ is executing short run-lengths of just 20 cycles. A good scheduling policy should switch out thread $A$ whenever thread $B$ is ready to run. This allows hiding the latency from as many of $B$'s references as possible.

**locking scenario:** Thread $A$ is executing with long run-lengths, while thread $B$ is attempting to obtain a lock, do a few critical operations, and release the lock. In order to minimize contention for the critical region, it is important for $B$ to hold the lock for as short a time as possible. Idealy thread $A$ should be switched out when thread $B$ is ready to run, giving $B$ priority when it is holding a lock.

**spinning scenario:** If thread $B$ is spinning while waiting for some event to happen, it should be given lower priority so that thread $A$, which is doing useful work, can make progress. In fact, it is essential that $A$ be given access to the processor since $B$ might be waiting on an event that will be caused by $A$.[6]

These scenarios all suggest that context switching must be done more often than just on cache misses. In fact, long run-lengths can be broken into several smaller and more

[6] Spinning is a bad idea on a multithreaded processor since the processor will usually have work that can be done by another thread. In Section 7.1.2 we will discuss the implementation of synchronization primatives that do not involve spinning.

uniform run-lengths to help improve their latency hiding capacity. Below are a number of basic scheduling policies that we studied alone and in combination with each other. These policies all switch on cache misses, but also context switch for the additional reasons specified by the policies.

**Basic Scheduling Policies:**

**spin-switch:** Spinning threads are switched out after every shared memory load instruction. This minimizes the number of execution cycles wasted by spinning threads.

**timeout(N):** Threads are forced to switch after they have held the processor for N cycles. (Tried with N ranging from 10 to 200.)

**lock-priority:** Threads holding a lock are given preemptive priority. This allows a thread to execute and exit a critical region as quickly as possible.

**new-priority:** Newly ready threads (those having just received a result from a remote reference) are given preemptive priority. The object is to give priority to those threads that are executing with short run-lengths.

**always-switch:** Threads are context switched after every shared memory load instruction regardless of whether it missed in the cache. This is a simple policy that gives all threads frequent access to the processor.

Table 5.4 shows the execution efficiencies under some of the scheduling policies that we studied. These simulations are for the **switch-on-miss** model, which will be discussed in the next section. We present these scheduling results first because the best scheduling policy found here will be used in the next section for the **switch-on-miss** simulations. Experimental parameters are specified in Table 5.5.

Overall, the best policy that we studied was one that combined **timeout(100)**, **lock-priority**, and **spin-switch**. This was selected as *best* based on averaging the execution efficiencies of all of the applications except `sieve` and `mp3d`. `Sieve` was excluded because it runs well without multithreading, and thus the scheduling policy is irrelevant when there is only one thread on a processor. `Mp3d` was excluded because we will see in Chapter 6 that its performance will likely be constrained by bandwidth rather than latency.

| Application | Multithreading | spin-switch | lock-priority + spin-switch | new-priority + spin-switch | always-switch | timeout(100) | timeout(100) + lock-priority + spin-switch |
|---|---|---|---|---|---|---|---|
| blkmat | M=3 | 75.3 | 75.6 | 76.6 | 77.3 | 76.2 | 76.2 |
| sor | M=4 | 79.7 | 79.7 | 84.4 | 89.6 | 88.8 | 88.8 |
| ugray | M=3 | 73.6 | 81.9 | 89.3 | 88.3 | 89.4 | 89.8 |
| water | M=3 | 89.1 | 92.8 | 92.8 | 92.1 | 93.4 | 93.5 |
| locus | M=3 | 74.5 | 75.3 | 84.8 | 85.6 | 85.6 | 89.5 |
| mp3d | M=11 | 84.5 | 84.5 | 83.7 | 92.4 | 84.6 | 84.6 |
| barnes | M=2 | 80.0 | 80.9 | 82.0 | 82.5 | 82.3 | 82.4 |
| Average (excluding mp3d) | | 78.7 | 81.0 | 85.0 | 85.9 | 86.0 | 86.7 |

Table 5.4: Execution efficiencies under various scheduling policies.

**Experiment: scheduling under switch-on-miss**

| Application | Processors | Multithreading |
|---|---|---|
| blkmat | 64 | 3 |
| sor | 16 | 4 |
| ugray | 32 | 3 |
| water | 29 | 3 |
| locus | 10 | 3 |
| mp3d | 32 | 11 |
| barnes | 32 | 2 |

- Latency = 200 cycles
- Context switch = 3 cycles if caused by a miss, 0 cycles if caused by the scheduling policy
- Scheduling = *experimental parameter*
- Each processor has a 64K byte cache with a 16 byte line size and 4 way set associativity.

Table 5.5: Experimental parameters for evaluating scheduling policies under **switch-on-miss**.

| **Experiment: switch-on-miss** | | | |
|---|---|---|---|
| Application | Processors | Multithreading | • Latency = 200 cycles<br>• Context switch = 3 cycles if caused by a miss, 0 cycles if caused by the scheduling policy<br>• Scheduling = timeout(100) + lock-priority + spin-switch<br>• Each processor has a 64K byte cache with a 16 byte line size and 4 way set associativity. |
| sieve | 128 | 1 | |
| blkmat | 64 | 1–4 | |
| sor | 16 | 1–4 | |
| ugray | 32 | 1–4 | |
| water | 29 | 1–3 | |
| locus | 10 | 1–3 | |
| mp3d | 32 | 1–11 | |
| barnes | 32 | 1–3 | |

Table 5.6: Experimental parameters for **switch-on-miss**.

Many other scheduling policies performed nearly as well as the chosen policy. In fact, simple policies such as **timeout(100)** or **always-switch** performed within 1% of the chosen policy on average. These policies address the three scenarios given above because they limit the interval in which a thread can dominate the processor. We thus conclude that choosing a particular scheduling policy is not critically important and can be based on what the hardware designer finds most convenient.

## 5.3 Switch-On-Miss

The experimental parameters used in our simulations of **switch-on-miss** are shown in Table 5.6. The context switch cost was 3 cycles if caused by a cache miss, but 0 cycles if forced by the scheduler because of some scheduling policy related decision such as a preemption or timeout. The differing context switch times depend upon whether the context switch decision is made early (scheduler) or late (cache miss) in the pipeline. This is explained in Chapter 8.

Figure 5.3 shows the execution efficiencies at various multithreading levels. The bars with $M = 1$ are the results that were presented in Section 5.1 for caching without multithreading. A few bars, such as $M = 3$ and $M = 4$ for `blkmat`, are unlabeled because there was not sufficient room to insert the labels. In all cases, these unlabeled bars correspond to the next sequential multithreading level.

At $M = 1$, most of the applications perform in the 60% to 70% efficiency range, and

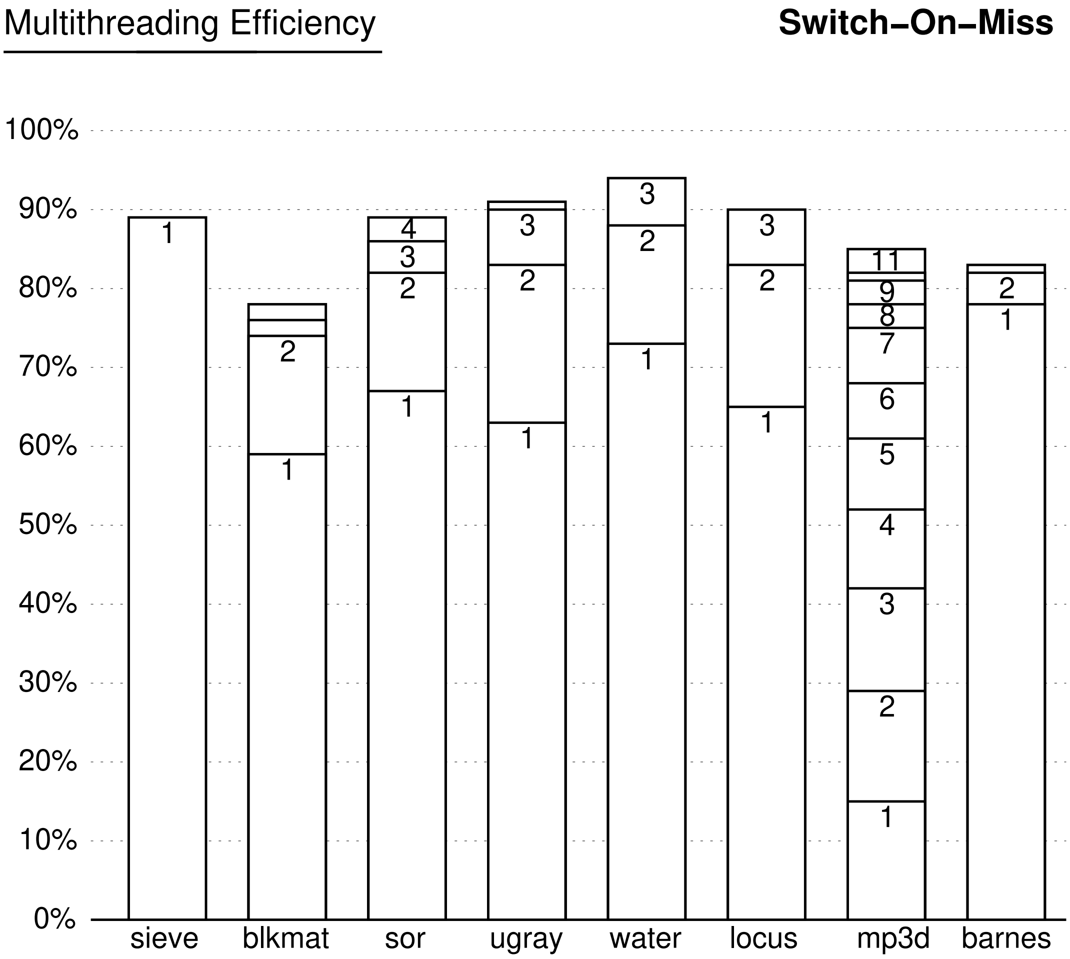



Figure 5.3: Multithreading levels and the efficiencies achieved under **switch-on-miss**.

the addition of multithreading raises the performance to the 80% to 90% range. Expressed in terms of relative performance (multithreaded performance/single threaded performance), multithreading provides a 30% to 40% performance increase for most applications. There are three exceptions. `Sieve` caches extremely well and thus has no use for multithreading. `Barnes` has a large performance loss due to synchronization, which is not helped by multithreading, and `mp3d` caches poorly and thus has room for and achieves much larger performance gains from multithreading.

The number of threads used for `sor` is small because of the sensitivity of its performance to the degree of parallelization. It partitions the 192 by 192 grid into as many square (or rectangular) regions as there are threads. Cache interactions occur just along the edges of these regions because the algorithm accesses only neighboring values in the grid. The cache hit rate is thus strongly affected by the size of the regions. To allow a fair comparison between **switch-on-miss** and **explicit-switch**, we kept the number of processors the same. This lets **switch-on-miss** receive the benefit of requiring fewer threads and thus having larger regions for a given problem size. In the configuration used here (P = 16, M = 4), /sor/ runs at 89% efficiency. With more processors and threads (P = 64, M = 4), and thus finer partioning, efficiency drops to 75%.

Compared to the results for multithreading without caching (from Chapter 4), the execution efficiencies vary from a few percent worse to 15% better, depending on the application. The big change is that since run-lengths are much longer with caching, not as many threads are needed, and the improvement due to multithreading is much less. For most of the applications, multithreading of 3 threads per processor is adequate to hide the 200 cycle latency.

## 5.4 Conditional-Switch

Grouping was very effective at improving the performance and decreasing the multithreading levels needed under **explicit-switch** compared to **switch-on-load**. We can apply the same idea to a caching system by treating the switch instructions conditionally. Under the **conditional-switch** model, if all of the references proceeding a switch instruction hit in the cache, the switch instruction is ignored, but if any of them miss, then the switch is taken in order to wait for the result(s). The potential benefit is that we can issue more than one reference per thread into the network before waiting for the results to return.

**Experiment: conditional-switch**

| Application | Processors | Multithreading |
|---|---|---|
| sieve | 128 | 1 |
| blkmat | 64 | 1–4 |
| sor | 16 | 1–4 |
| ugray | 32 | 1–4 |
| water | 29 | 1–3 |
| locus | 10 | 1–4 |
| mp3d | 32 | 1–9 |
| barnes | 32 | 1–3 |

- Latency = 200 cycles
- Context switch = 1 to 3 cycles
- Scheduling = timeout(100) + lock-priority + spin-switch
- Each processor has a 64K byte cache with a 16 byte line size and 4 way set associativity.

Table 5.7: Experimental parameters for **conditional-switch**.

The experimental parameters for **conditional-switch** are shown in Table 5.7. They are the same as those for **switch-on-miss** except for the different multithreading model and different context switch timing assumptions. The context switch cost varies from 1 to 3 cycles depending on when the cache miss occurs relative to the context switch instruction. If the cache miss has already occurred when the switch instruction enters the pipeline, the context switch can be done immediately as it was for **explicit-switch**. Otherwise, the context switch will occur deeper in the pipeline as it did for **switch-on-miss**.

Figure 5.4 shows the performance of the applications under **conditional-switch** multithreading. These simulations were run using grouping only within basic blocks, since we do not have a compiler that can do inter-block grouping. At this level of grouping, all of the applications have equivalent or lower performance than under **switch-on-miss**, except for `mp3d`. `Mp3d` is an exception because it does not cache well and thus retains some of the behavior of an uncached system.

The lower performance indicates that grouping is not useful in conjunction with caching. This occurs because grouping is beneficial only when more than one reference is sent into the network before a context switch. If we look at a group of references when the cache is working well, usually all or most of the references will hit in the cache, and only in the rare case when two (or more) references both miss, will grouping be beneficial. In fact, even this overstates the benefit of grouping since sometimes the two accesses may be to elements of the same cache line, and thus the first access would have brought both values in anyways.

Any benefit of grouping is counterbalanced against the extra cost of the added

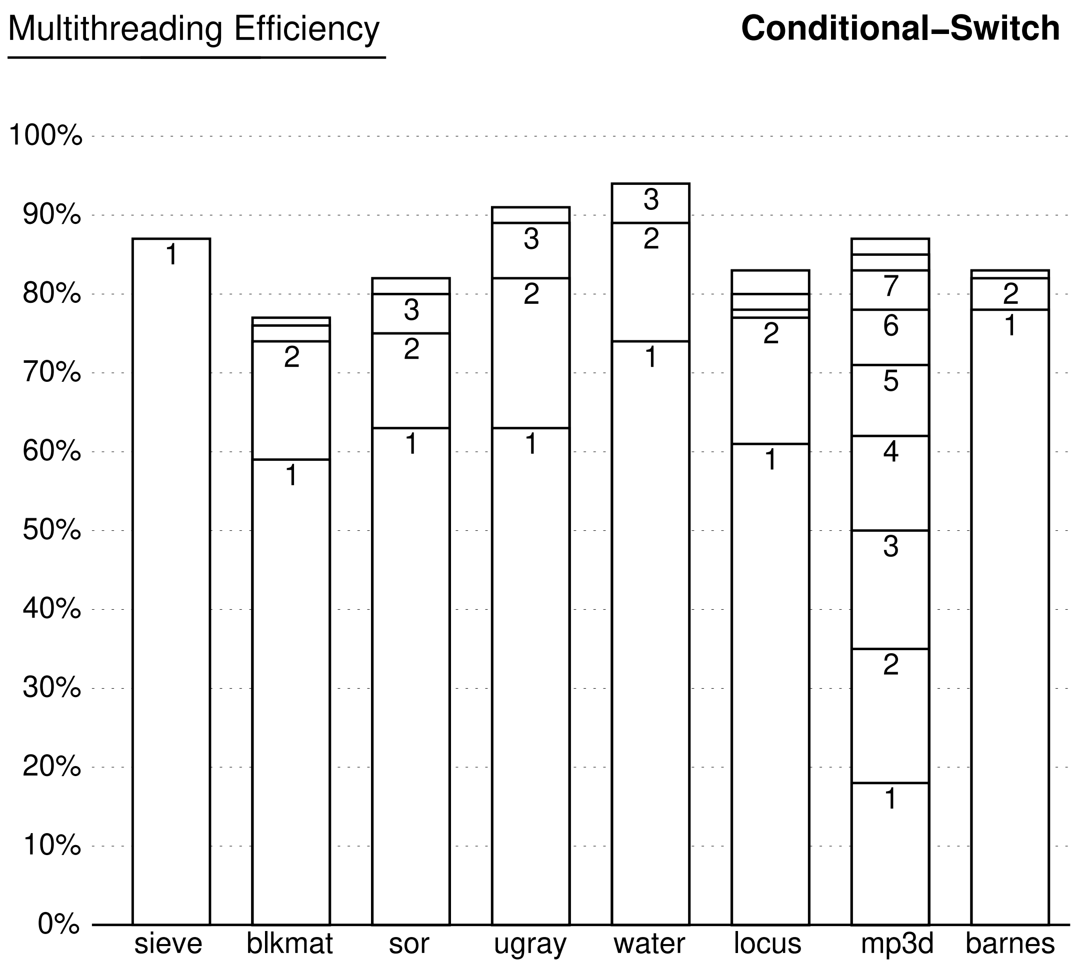



Figure 5.4: Multithreading levels and the efficiencies achieved under **conditional-switch**.

context switch instructions. These extra instructions take a cycle in the execution stream regardless of whether they are useful or not.

## 5.5 Conclusions

Caching is effective for most of our applications. We observed miss rates ranging from 1% to 4%. These low miss rates mean that threads execute for longer intervals before context switching and thus fewer threads will be needed to hide the latency.

However, sometimes these long execution intervals can cause performance problems by letting one thread hold the processor and thereby block other threads from executing. This can be dealt with by adding a timeout or other mechanism to the scheduling policy.

Our simulations show that a machine without multithreading can obtain efficiencies of 60% to 70% with a latency of 200 cycles, and that a machine with **switch-on-miss** multithreading using 3 threads per processor can boost these efficiencies to 80% to 90%.

Finally, our simulations of the **conditional-switch** model show that grouping is not beneficial in conjunction with caching.

# Chapter 6

# Limited Bandwidth

In the previous chapters we have shown that the long latencies of the communication network can be tolerated by using multithreading techniques. In this chapter we look at the other main characteristic of a communication network: bandwidth.

Where as long latencies are inevitable because of the large number of processors and memories that must be connected together, the bandwidth capacity of a network can be increased by spending more money and adding more wires and/or switches. Unfortunately, as machines grow, the network becomes a larger and larger fraction of the total system hardware. For example on indirect networks such as butterflys and fat-trees[Lei85], $O(p \log p)$ routing nodes are used to connect $p$ processors. For direct networks, the number of routing nodes is the same as the number of processors, but if you count pins and wires, the amount of hardware increases for direct networks as well. On a hypercube, the degree of the routing nodes increases as $O(\log p)$. On a 2-D mesh, the width of the channels must grow as $O(\sqrt{p})$ if a fixed bisection bandwidth/processor is to be maintained. The bottom line is that for a large machine, the network will be expensive, and therefore we need to understand and minimize the bandwidth demands put upon it.

In this chapter we present the bandwidth needs of our benchmark application suite under the **explicit-switch** and **switch-on-miss** multithreading models. Our results will show that caching substantially reduces the the network bandwidth needed. We then look more closely at the traffic patterns of **switch-on-miss** systems. The traffic on these systems will be bursty and thus some execution periods will need more network bandwidth that others. We measure this burstiness and use the results along with a performance model to suggest the level of bandwidth that should be suplied by the network.

| **Experiment: remote memory bandwidth needs of explicit-switch** | | |
|---|---|---|
| Application | Processors | Multithreading |
| sieve | 32 | 5 |
| blkmat | 64 | 3 |
| sor | 16 | 9 |
| ugray | 12 | 10 |
| water | 20 | 5 |
| locus | 8 | 7 |
| mp3d | 32 | 11 |
| barnes | 16 | 5 |

- Latency = 200 cycles
- Context switch = 1 cycle
- Scheduling = round robin
- Inter-block grouping estimates as in Section Section-IBG[1]

Table 6.1: Experimental parameters for measuring the remote memory bandwidth needs of the applications under **explicit-switch**.

## 6.1 Bandwidth Requirement

The bandwidth which an application uses depends upon a number of factors. First, the application may be either computationally or communication intensive. Second, if the machine provides caching, much of the potential traffic may get filtered out by the cache. And third, if the processor is multithreaded, the higher processor utilization and thus higher computational rate will increase the bandwidth requirement.

### 6.1.1 Bandwidth Requirement Without Caching

We measured the bandwidth requirements of the applications by summing the sizes of all messages sent through the network. This gave us the total traffic used by an application. We then normalized this to bits/cycle/processor by dividing the total traffic by the execution time and by the number of processors. We call this the *remote memory bandwidth.*

Table 6.1 lists the simulation parameters. We measured the bandwidths of **explicit-switch** with inter-block grouping, which was the best performing multithreading model (without caching). The bandwidths were computed based on the message sizes shown in Figure 6.1. These messages are used for sending loads and stores and for returning their results. The first field in a message is its destination memory module or processor.

[1]For the inter-block grouping estimates we used a one line cache for each thread. This affected the grouping but not the bandwidth results. The bandwidth was calculated as if all messages were sent into the network and no caching was present.

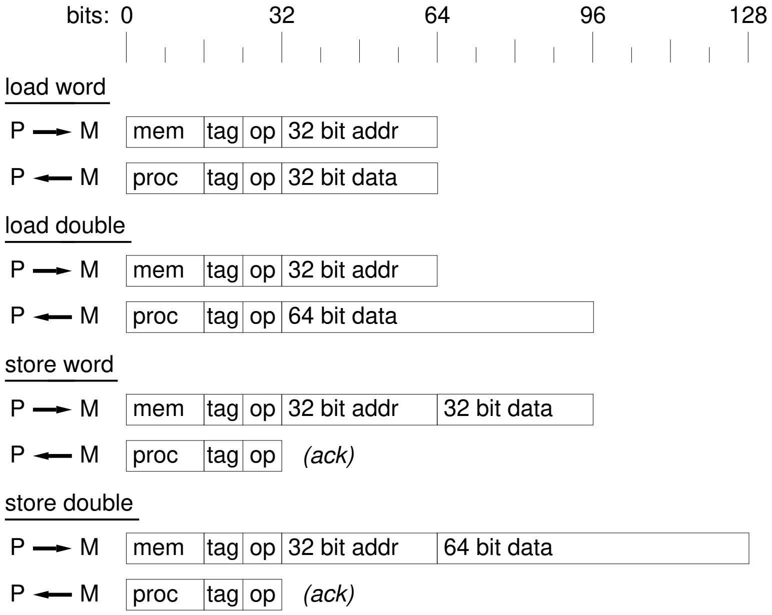



Figure 6.1: Message sizes for remote references to shared memory.

Next is an 8 bit tag field that is used to identify results as they are returned[2]. Then is an 8 bit opcode that specifies the operation type and message size. The last field(s) is either the address being referenced, the data returned, or the address and data for a write.

These messages sizes are at the small end of the spectrum of possible implementations. For instance, we have assumed that the only routing information needed is the number of the destination memory bank or processor, and that the return address can be generated as the message is routed[GGK+82]. Also we have used 32 bit addresses, whereas a large parallel machine will likely support a larger address space. To apply our simulation results to a machine using larger messages, our bandwidth results should be scaled up proportionally to the increase in message sizes.

Table 6.2 shows the remote memory bandwidth results. The bandwidths vary considerably by application, and range as high as 30 bits/cycle/proc to as low as 1.44 bits/cycle/proc. For comparison, Table 6.3 shows the bisection bandwidths of proposed and existing machines. These bandwidths are for machines scaled to 1024 processor and are taken from Figure 1.4 in Chapter 1.

At first glance, our measurements of remote memory bandwidth in Table 6.2 may

[2]See Section 8.1.2.