Architectural Requirements for In-Pipeline CCMs

J. Gregory Steffan
Reconfigurable Computing Seminar, 15-828
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
steffan@cs.cmu.edu

May 7, 1998

Abstract
The incorporation of a custom computing machine (CCM) into the pipeline of a superscalar processor has been shown to allow significant gains in performance for general-purpose codes [1,2,3]. This paper describes the architectural features necessary for such an in-pipeline CCM to achieve maximum potential performance gains when all mapping is done automatically by the compiler. We find that a CCM with two inputs and two outputs that computes a result in one cycle captures the performance benefits of the maximal architecture. We also find that the ability to use four unique configurations within a small number of cycles can be very beneficial.

1 Introduction

Previous studies have shown that the incorporation of a custom computing machine (CCM) into the pipeline of a superscalar processor can result in significant performance gains [1,2,3]. However, these studies have assumed an architecture for the CCM and then measured the performance of mapping to it whenever beneficial. This study takes a different approach: while making aggressive assumptions about the capabilities of a CCM, we measure the performance benefits of CCMs of varying architectural parameters. More complex CCMs undoubtedly result in improved performance, but at a certain point the increased complexity of the architecture offers diminishing returns. The goal of this work is to find this point in the CCM design space.

1.1 What is an In-Pipeline CCM?

An in-pipeline CCM is a small, reconfigurable device that resides in the processor pipeline, as shown in Figure 1. It is basically treated like a regular functional unit: a special CCM instruction specifying the configuration, register inputs and outputs is issued to the CCM. More details of the CCM instruction mechanism are given in [3, 5]. If the CCM is not currently configured to perform the desired computation, then the configuration must be loaded from memory or special configuration storage.
1.2 How Do CCMs Improve Performance?

In this study, we assume that a CCM is some form of small FPGA. Rather than attempting to map complex software kernels to the CCM, we try to improve performance in one of the following ways:

**Parallelism:** do multiple, independent things at the same time;

**Combining Instructions:** compute the result of several instructions in one instruction (e.g., a shift followed by an add);

**Jump Tables:** some control flow could be optimized by computing the target address of a multi-way branch rather than traversing a series of if-then-else clauses [3, 5].

1.3 Previous Architectures

The PRISC architecture [3, 4] assumes 2 inputs, 1 output, 1 cycle for computation, and has support for one configuration at a time. They used a tool which could automatically extract CCM computation from the object files of benchmarks, and achieved decent speedup for several SPEC92 [2] benchmarks.

The Chimaera architecture [5] assumes nine inputs, one output (during a given cycle), one cycle for computation and has support for several simultaneous configurations. Although computation effectively occurs within 1 cycle, chimaera also supports multi-cycle computations by speculatively computing results every cycle but only writing the result one cycle after being called. In the cited paper the Chimaera is mapped to by hand, resulting in significant speedups.

These studies both assume a CCM architecture and then measure the performance gains of using it in general purpose applications. In contrast, this study looks for the available CCM computations in general purpose applications and attempts to discover the benefits of different architectures.

The remainder of the paper is organized as follows. Section ?? gives an overview of the experimental methodology used in this study, and Section 3 describes the compiler. We investigate the CCM architecture required to achieve potential performance improvements in Section 4, and measure the effects of different configuration management strategies in Section ???. Finally, we conclude
2 Methodology Overview

Experiments for this study were performed on the Spec95 [1] integer benchmarks. It is worth noting that the Spec95 integer benchmarks are significantly more difficult than the Spec92 [2] integer benchmarks with respect to finding computation suitable for execution on a CCM. Our methodology proceeds as shown in Figure 2.

The source for each benchmark is fed to ccomify, a collection of passes built in the SUIF [6] compiler system. The compilation process is described in detail in Section 3; for now, we focus on a high level description. Several SUIF passes search for available CCM computation, and then instrument the desired locations in the code with blank procedure calls. Figure 3 demonstrates how a blank procedure call is used to represent a CCM configuration: the resulting CCM expression will have one input, one output, take one cycle to compute, and will save four cycles of execution time. The parameters of the procedure call indicate these values. Not shown are two other parameters: an integer denoting the parallelism of loop configurations; and a configuration identifier, used to perform configuration management studies. The back-end of SUIF is then used to produce C-source code, augmented with the CCM procedure calls, which is then compiled using a standard compiler (cc) to produce an instrumented binary.

The instrumented binary is passed to the simulator which is based on the MINT [7] emulator. MINT allows instrumentation to be inserted before or after any instruction, basic-block, or procedure, and since it is an emulator the instrumentation will not affect the cycle times of the running code. The MINT interface makes it easy to implement a single-CPI simulator that assumes every instruction takes one cycle to execute.
The simulator, called \texttt{ccmsim}, instruments every call to a CCM procedure call. The procedures parameters are noted, and the cycle savings for that particular CCM configuration are deducted from the cycle count. The overhead of the calls that were inserted is also deducted. Furthermore, the simulator has support to vary the configuration miss latency, and the number of simultaneous configurations that may reside in the CCM.

3 The Compiler

The CCM analysis is performed by the following series of \textit{SUIF} passes:

1. All MBR instructions are annotated as CCM-computable.

2. Simple width analysis is performed. This involves looking for a bitwise-\texttt{and} instruction where one operand is a small constant, or a convert instruction with a narrow destination. If either are found, then an annotation which denotes “narrowness” is propagated down the expression tree. This way, all instructions in an expression tree that eventually produce a narrow result are marked as being narrow.

3. Expression analysis is performed. This is described in detail in Section 3.1.

4. Loop analysis is performed: here we find expressions with narrow-width results that reside within the body of a for loop. Since the results are narrow, several of these expressions may be computed in parallel.\footnote{For simplicity, we make the aggressive assumption that there are no data dependences between expressions. We see in Section 4 that this does not overly inflate performance.}

5. Interference analysis is performed. This involves deciding which CCMs to utilize in order to minimize configuration interference. Two versions of this were implemented: choosing the best CCM within a given loop nest; and choosing the best CCM within a given procedure. Two CCMs are compared by favoring loop configurations over others, and by choosing the CCM with the largest cycle savings otherwise. Results given in Section 4 demonstrate that interprocedural analysis is necessary for proper interference analysis.

6. Blank procedure calls are inserted for all chosen CCMs. The procedure parameters indicate the number of inputs, outputs, cycle savings, computation depth, parallelism, and an identifier for the CCM (the identifier is used for configuration interference studies).

3.1 What Makes an Expression CCM-Computable?

An expression is considered to be CCM-computable by the following heuristics:
Table 1: Static instance counts of CCM configurations.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Expr</th>
<th>MBR</th>
<th>Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>ccl</td>
<td>1183</td>
<td>180</td>
<td>721</td>
</tr>
<tr>
<td>compressS6</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>go</td>
<td>51</td>
<td>28</td>
<td>8</td>
</tr>
<tr>
<td>li</td>
<td>7</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>m88ksim</td>
<td>540</td>
<td>31</td>
<td>38</td>
</tr>
<tr>
<td>perf</td>
<td>227</td>
<td>37</td>
<td>56</td>
</tr>
<tr>
<td>vortex</td>
<td>91</td>
<td>35</td>
<td>35</td>
</tr>
</tbody>
</table>

1. The following instructions are considered to be “free” (i.e., they do not take any cycle time to compute): cvt, not, cpy, neg, ldc. This is a reasonable assumption, since converts and copies are just wiring, logical not and negate are easy, and loaded constants will be part of the CCM configuration.

2. The following instructions are also considered to be free if the second source operand is a constant: asr, lsr, lsl, rot. If the second source operand is constant, then the instruction is reduced to wiring.

3. All other operations are assumed to be one cycle to compute if one operand is a constant or both operands are narrow. One of the two previous conditions must be met for the operation to be annotated as CCM-computable.

Each expression tree is traversed and the largest possible CCM-computable subexpression is found, given the limits to the number of inputs, outputs, and computation time. SUIF performs local constant folding and propagation, so no trivial, extraneous computation remains.

### 3.2 Static Results

The compiler found a wide variety of CCM configurations in the benchmarks. Table 1 shows the static instances of each type of configuration in each benchmark. compress has relatively little source code, and thus has few configurations. m88ksim does quite well, and cc1 (gcc) has many configurations. Expression configurations are dominant, and MBR and loop configurations are found less frequently.

The benchmark cc1 has the widest variety of configurations, so we show the composition of its configurations is given in Table 2. For each configuration type, we see the architectural demands as well as the static count. A majority of configurations have two or fewer inputs, and also two or fewer outputs. Computation depth is never more than two cycles. These configurations are representative of those found in the other benchmarks, in that they tend to have few inputs and outputs, and one cycle computation time. For this reason, the baseline, maximal architecture was chosen to have a computation time of two, eight inputs, and eight outputs.
Table 2: CCM Configurations in cc1.

<table>
<thead>
<tr>
<th>CCM Type</th>
<th>Computation Depth</th>
<th>Number Inputs</th>
<th>Number Outputs</th>
<th>Parallelism</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBR</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>180</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1150</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Loop</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>180</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>8</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>46</td>
</tr>
</tbody>
</table>

Table 3: Source file coverage.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total</th>
<th>Std</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress95</td>
<td>2</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>go</td>
<td>17</td>
<td>17</td>
<td>100</td>
</tr>
<tr>
<td>m88ksim</td>
<td>102</td>
<td>99</td>
<td>97</td>
</tr>
<tr>
<td>perl</td>
<td>20</td>
<td>13</td>
<td>95</td>
</tr>
<tr>
<td>vortex</td>
<td>86</td>
<td>66</td>
<td>100</td>
</tr>
</tbody>
</table>

Unfortunately, two of the benchmarks did not make it all the way through the system: cc1 and li. Furthermore, not all of the source files for some benchmarks were successfully instrumented. Table 3 shows the percentage of source files that were successfully instrumented. For perl, this partial coverage hinders the results significantly, since important source files like the regular expression matching code were not instrumented.

4 Architecture

To investigate the maximum performance benefit of the available CCM computation we measured speedup for each benchmark with the configuration miss latency being zero. The maximal architecture targeted has a computation time of two cycles, eight inputs, and eight outputs. The results, given in Figure 4, show that m88ksim speeds up near 18 percent, while the other benchmarks have speedups of less than 6 percent.

Figure 5 shows the composition of speedup for each benchmark by configuration type. Most of the speedup in compress95 is due to loop configurations, go has a mixture of all three types, and the other benchmarks mostly benefit from MBR configurations.

Now we will measure the decrease in potential speedup for each benchmark when the architectural features are incrementally limited. We start by limiting the number of computation cycles from two to one, as shown in Figure 6. Since multi-cycle configurations are not that prevalent, all of the baseline speedup is still achieved for every benchmark except compress95, which only suffers
Figure 4: Speedup (given as a percentage) assuming a computation time of two cycles, eight inputs, eight outputs, and configuration miss latency of zero. This is the baseline cycle savings.

Figure 5: Breakdown of the baseline configuration into percentage of overall cycle savings for the different configuration types.

slightly.

Next we decrease the number of inputs from eight to one, as shown in Figure 7. Again, since configurations with more than one input are infrequent, the speedups for configurations limited to fewer than eight inputs are still near maximal. There is a noticeable decrease in performance when the number of inputs is limited from two to one, suggesting that two inputs are useful.

By decreasing the number of outputs, as shown in Figure 8, we see that speedups are maintained except when limited to one output. Since the only configurations that utilize more than one output are the loop configurations, this indicates that loop configurations with more than two inputs do not contribute significantly to speedup. This is most likely due to the lack of computation that is more narrow than 16 bits, since we do not parallelize beyond a width of 32 bits total (i.e., an expression of width 16 bits has a parallelism of two in this scheme, since we may compute two such expressions in a width of 32). These results suggest that two outputs are sufficient.

The previous three experiments indicate that a minimal architecture, given our method of finding CCM computation, consists of a CCM with two inputs, two outputs and one cycle for computation. We measure the performance of this architecture in contrast with the baseline in Figure 9, where it is evident that the minimal architecture captures all of the available performance benefit with the exception of compress95, where it captures 97%.
Figure 6: Percentage of the baseline cycle savings when limited to a computation time of one cycle.

Figure 7: Percentage of the baseline cycle savings when varying the number of inputs.

5 Configuration Caching

Since CCMs can take a long time to reconfigure (100 to 500 cycles), it is important that the number of times that the CCM must be reconfigured is not too great, or at least not enough to negate the performance benefits of using the CCM. The compiler inserts calls to unique CCM configurations in various locations in the code, but it is crucial that this is done with reconfiguration in mind. If more than one configuration is used inside a tight loop, then the configurations will interfere, or cause many reconfigurations to occur since the desired configuration will never be loaded. We will call each reconfiguration a configuration miss.

There are several ways to minimize the number of configuration misses. The compiler can make wiser decisions about when to use the CCM, or the CCM itself can have support for multiple simultaneous configurations or at least quick reconfiguration time for a few configurations. In this section we investigate the effectiveness of these techniques. All of the experiments in this section were performed using the minimal architecture described by the previous section (two inputs, two outputs, and one cycle computation time). For simplicity, we also assume that every configuration chosen statically by the compiler is unique. To test whether two configurations were actually unique would require storing the implemented expression tree with each CCM annotation.

Figure 10 shows the miss rate for a varying number of simultaneous configurations. We have
Figure 8: Percentage of the baseline cycle savings when varying the number of outputs.

Figure 9: Percentage of the baseline cycle savings for the minimal architecture: one cycle computation time, two inputs and two outputs.

modeled this experiment as a configuration cache, where the least recently used configuration is replaced on a configuration miss. We observe that m88ksim has a very large miss rate, as does perl. When we add support for two simultaneous configurations, compress95 almost never misses since it mainly uses the same two configurations inside a tight loop. The other benchmarks, with the exception of m88ksim, also benefit from the extra configuration. When up to four simultaneously configurations are supported, the miss rates for all of the benchmarks, again with the exception of m88ksim, improve significantly.

m88ksim is a pathologically bad case with respect to reconfiguration. It is coded as a small loop which rotates through lots of procedure calls, but none of the procedure calls contain loops of any significance and are very short. This means that any configurations chosen in the call graph below the loop will interfere with each other. Since the compiler has chosen more than four configurations, they interfere “perfectly” and cause many misses. It is therefore desirable to have the compiler make more intelligent decisions with respect to managing interference.

Figure 11 shows the number of cycles between reconfiguration for the minimal architecture. For each benchmark, the bars represent the percentage of reconfigurations that occurred within less than 100 cycles, between 100 and 200 cycles, and so on up to greater than 900 cycles. compress95,
Figure 10: Configuration miss rate for a varying number simultaneous configurations.

Figure 11: Number of cycles between reconfiguration for the minimal architecture. Times vary from less than 100 cycles to greater than 900 cycles.

$m88ksim$ and $vortex$ have an unacceptable number of reconfigurations that occur with less than 100 cycles between them. The previous experiment demonstrated that having support for simultaneous configurations can solve the problem for $compress95$, but not for $m88ksim$. We therefore must try to manage interference with the compiler as well.

Figure 12 shows the same measurements when the compiler has chosen the best configuration within every loop body. If there are several nested loop bodies, only one configuration is chosen for the entire nest. Note that interprocedural analysis is not performed. The best configuration is defined as the one with the largest cycle savings. With the exception of $go$, there is almost no improvement when compared with Figure 11.

Figure 13 shows the experiment repeated, except the compiler now chooses only the best configuration for each procedure—this is the fewest number of configurations that can be chosen without performing inter-procedural analysis. Again, we see almost no improvement, and even some degradation such as in $perl$.\footnote{This degradation is probably due to configurations that used to achieve high hit rates no longer being chosen.}

Figure 14 summarizes the previous three experiments by showing the improvement in miss rate for the different interference-management strategies implemented in the compiler. Only having one configuration per procedure improves the miss rate for $m88ksim$, but this is misleading: we have
Figure 12: Number of cycles between reconfiguration when the best CCM per loop body is chosen. Times vary from less than 100 cycles to greater than 900 cycles.

Figure 13: Number of cycles between reconfiguration when the best CCM per procedure is chosen. Times vary from less than 100 cycles to greater than 900 cycles.

also decreased the number of configurations used and thus the overall speedup.

Interference-management is thus a cost-benefit trade-off, where the cost is the latency of reconfiguration, and the benefit is the cycle-savings of using the CCM. If the compiler has knowledge of the underlying configuration support (i.e., the number of simultaneous configurations and replacement strategy), it is possible for it to make wise decisions when scheduling CCM configurations. Interprocedural analysis is definitely required for proper configuration-management, since it would be desirable to only have the proper number of configurations within a tight loop, even if the loop calls other procedures. Profile information would also be very beneficial.

6 Conclusions

The results from this study indicate that an in-pipeline CCM architecture should have at least two inputs and at least two outputs in order to achieve the potential performance benefits inherent in general-purpose applications. We also found that multi-cycle computations are not prevalent, so that CCMs without state—where all results are computed in a single cycle—are sufficient.

Experiments suggest that having four configurations that can be loaded quickly will improve
Figure 14: Configuration miss rate for three interference-management strategies: implement all configurations (ALL); only implement the best configuration within a given loop nest (1 Per Loop); only implement the best configuration within a given procedure (1 Per Proc).

performance, as well as ease the difficulty of configuration management. It is also evident that interprocedural analysis is a necessary part of properly handling configuration interference.

References


