Exploiting Operation Level Parallelism through Dynamically Reconfigurable Datapaths

Zhining Huang, Sharad Malik
Department of Electrical Engineering, Princeton University
Princeton, NJ 08544, USA
{znhuang, sharad}@ee.princeton.edu

ABSTRACT

Increasing non-recurring engineering (NRE) and mask costs are making it harder to turn to hardwired Application Specific Integrated Circuit (ASIC) solutions for high performance applications [12]. The volume required to amortize these high costs has been increasing, making it increasingly expensive to afford ASIC solutions for medium volume products. This has led to designers seeking programmable solutions of varying sorts using these so-called programmable platforms. These programmable platforms span a large range from bit-level programmable Field Programmable Gate Arrays (FPGAs), to word-level programmable application-specific, and in some cases even general-purpose processors. The programmability comes with a power and performance overhead. Attempts to reduce this overhead typically involve making some core hardwired ASIC like logic blocks accessible to the programmable elements. This paper presents one such hybrid solution in this space – a relatively simple processor with a dynamically reconfigurable datapath acting as an accelerating co-processor. This datapath consists of hardwired function units and reconfigurable interconnect. We present a methodology for the design of these solutions and illustrate it with two complete case studies: an MPEG 2 coder, and a GSM coder, to show how significant speedups can be obtained using relatively little hardware. The co-processor can be viewed as a VLIW processor with a single instruction per kernel loop. We compare the efficiency of exploiting the operation level parallelism using classic VLIW processors and this proposed class of dynamically configurable co-processors. This work is part of the MESCAL project, which is geared towards developing design environments for the development of application specific platforms.

Categories and Subject Descriptors

J.6 [**COMPUTER-AIDED ENGINEERING**]: *Computer-aided design (CAD)*

General Terms

Design, Performance

1. INTRODUCTION

Deep sub-micron issues of interconnect delay and signal integrity have significantly increased the design costs of ASICs – both due to the higher engineering costs resulting from longer design cycles, as well as due to the increasing cost of design tools. This increase in NRE costs has pushed up the breakeven volume point at which the per-device cost for the manufactured product is viable. As a consequence, for many products below this break-even volume mark there is a desperate need to satisfy the application performance requirements using specialized programmable solutions. These programmable solutions are increasingly referred to as programmable platforms, and their design (and use) is an exercise in trying to match application concurrency with architectural concurrency.

There is a vast range of these platforms, from bit-level programmable fine and coarse-grained FPGAs, to word-level programmable application specific processors. The application specific processors themselves span a fairly large range – from specialized VLIW processors with complete instruction sets tailored to the application domain [14], to simple RISC processors with a few specialized instructions implemented using hardwired logic blocks [15]. A relatively new class of platforms consists of simple processors with programmable co-processors, which are essentially accelerating datapaths with very little, if any, control [8, 16].

In this paper we present a solution in the last category. We introduce a dynamically reconfigurable co-processor with fixed hardware blocks and programmable interconnect and show how this can be used to accelerate the main computation loops (kernels) of compute intensive applications. The rest of the application runs on a relatively simple master processor. Specifically we provide a complete methodology and algorithm for the design of such a co-processor for a given application as well as the mapping of the kernel loops onto this co-processor.

This is illustrated through two complete non-trivial case studies, an MPEG2 encoder and a GSM encoder where we show significant speedups using relatively little hardware.

Interestingly, this co-processor can be viewed as a VLIW processor with a single instruction per kernel loop. In our effort to understand the matching of application concurrency to architectural concurrency, we compare the efficiency of exploiting the operation level parallelism using classic VLIW processors and this proposed class of dynamically reconfigurable co-processors with some interesting observations.

2. PREVIOUS RESEARCH

There have been several research efforts as well as commercial products that have tried to explore the use of configurable logic as acceleration co-processors.

Early research on reconfigurable computing focused on configurable logic in the form of fine-grained FPGAs. However, FPGAs failed to become a mainstream computing technology for complete applications because of reasons such as bit level logic granularity, hardware constraints and high configuration time [5]. Subsequently we saw the emergence of tightly coupled reconfigurable co-processor designs, for example, GARP [1] and RaPiD [6]. In these designs, kernels from applications are mapped to the co-processor for acceleration. However, as these co-processors still consisted of fine-grained architectures, the problems of reconfiguration bit size, hardware resource constraints, and slow clock rate were not fully solved. Coarse-

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.

grained reconfigurable computing, such as used in PipeRench [5] and Pleiades [7] differed in that they used coarser blocks of logic in the reconfigurable fabric, resulting in faster clock speeds and the ability to implement larger applications in the multimedia and communication domains. However, PipeRench is not targeted to specific applications and cannot be customized for specific kernel loops. Pleiades is a design for a specific application domain targeting low power. It has a hierarchical mesh interconnect between logic macros (ALUs, MEMs, etc). The interconnect is reconfigured for different kernels to enable the reuse of logic macros.

Recently, commercial products have emerged that include the ability to dynamically reconfigure the programmable logic during a single application. The Chameleon chip has a 32bit embedded processor with a dynamically reconfigurable fabric. Up to 84 datapath units and 24 multipliers reside in the fabric. The interconnection between them is dynamically reconfigurable. The Chameleon chip has a fixed architecture targeting communication applications. Kernels in applications are mapped to the reconfigurable fabric through hand written Verilog descriptions. The reconfigurable fabric is significantly general to permit a wide application for communication applications. This device is an example of the dynamically reconfigurable co-processors that our methodology targets. We describe how such a device can be designed and programmed for a given application (domain) by providing a complete methodology. This builds on our previous work on this, which focused on a specific algorithm that is part of this methodology. None of the previous efforts described here have presented a methodology for the design of application specific co-processors that use dynamically reconfigurable coarse-grained logic.

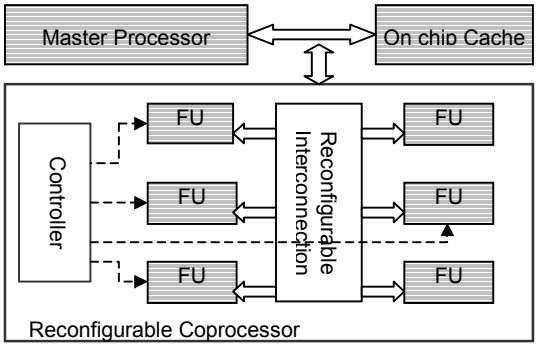

Figure 1. Architectural Model

3. METHODOLOGY OVERVIEW

The architectural model studied in this paper consists of a master processor and a reconfigurable co-processor as illustrated in Figure 1. The co-processor has the same access to the memory sub-system as the master processor. The co-processor mainly consists of a reconfigurable datapath and some control logic. The controller works as a state machine and controls the start and end of the datapath execution. The reconfigurable datapath has fixed ASIC-like function units and reconfigurable interconnections. This paper mainly describes the methodology of designing the reconfigurable datapath for an application, given its high-level language description.

The first step in the methodology is the identification of the computationally intensive loops in the application. For each such kernel loop a custom datapath is designed to maximize the parallelism that can be exploited. Datapaths designed for different loops in an application are then combined into a single reconfigurable datapath co-processor. The co-processor switches between the individual datapaths through changing the connections between the function units using programmable interconnect. The interconnect "program" bits for a loop datapath are referred to as the context of the loop. If the size of this program is small (as we will demonstrate) and the number of contexts is small (as we will also show), then the contexts can all be stored locally in the co-processor and need not be brought in from external memory. It also enables rapid switching of contexts – a switch can be accomplished in a single cycle.

4. DATAPATHS FOR KERNEL LOOPS

This section describes the first step in this methodology - how to generate the datapaths for kernel loops given high-level language descriptions of an application.

4.1 Extract Kernel Loops

We use the IMPACT compiler as the front end to do pre-processing, including performance profiling and loop detection. C codes are used as the description of applications and as the inputs to the front-end compiler. As we only map innermost loops to custom datapaths in the co-processor at this stage, the result of extracting kernel loops is a list of the most-executed innermost loops in Lcode (a meta assembly language), which is the intermediate representation of the IMPACT compiler.

In order to map kernel loops to customized pipelined datapaths, we need to do data dependence analysis to derive the following: register live-in set of the loop body; register live-out set of the loop body; data dependence between instructions within loop iterations; data dependence between instructions in different loop iterations.

The live-in and live-out sets are used during the switch between the master processor and the co-processor. When the execution of the application hits the loop, the general processor switches the execution to the co-processor. During the switch, the values of registers in the live-in set are copied into the co-processor. The live-out set contains registers whose value has changed during the loop execution and will be used later. After the loop execution, the co-processor switches execution back to the master processor and writes back the live-out register values.

The data dependence information helps in constructing the datapath for the loop body.

4.2 Direct Mapping of Kernel Loop Datapath

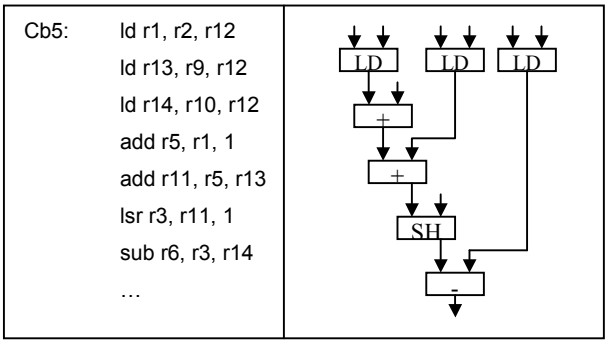

Figure 2. Direct mapping from IR to hardware

Given the Lcode intermediate representation for a kernel loop, our custom datapath design for that loop proceeds by starting with a direct hardware mapping from the Lcode to hardware blocks. Direct mapping here means that one instruction in software corresponds to one function unit in the hardware. By connecting all the function units in the hardware, we form the datapath for a loop body as illustrated in Figure 2.

Each function unit in the hardware executes an instruction in the software. Since the data goes through the datapath, no write back to the register file is needed. This further implies no register bandwidth limitation within loops. The advantage of direct mapping is the simplicity of the methodology, which makes it easy to implement automatically.

4.3 Branch Condition Transforms

If there is more than one basic block in the loop body, the two sides of branch instructions result in different datapaths, which are then merged using a multiplexer.

4.4 Pipelining the Execution

As the first step in pipelining the execution, we need to assign the function units into different pipe stages in the datapath. In order to do so, we insert registers in the datapath.

The scheduling of the pipeline stages for the datapath maximizes the parallelism of the loop body within the iteration. The scheduling of execution of consecutive loop iterations maximizes the parallelism between loop iterations. If there is no data dependence between loop iterations, at each clock cycle, we can fetch and begin to execute the next loop iteration until we finish all the iterations. If there is data dependence between loop iterations, we look at the pipeline stages of the instructions that cause the dependence to decide whether a delay or a bypass is needed. There are two kinds of data dependences, one is dependence from registers, and the other one is dependence from memory operations. The two kinds of dependences have different properties.

For register dependence, the register value is written in the first loop iteration and is used in the next one. The dependence always exists between adjacent iterations. Suppose the register value is written at pipeline stage i and is used at pipeline stage j. If $i \leq j$, the register value is always generated before it is used because of the pipelined execution. The second iteration is always one step behind the first one during the pipeline execution. So all we need is a bypass from the function unit that generates the register value to the function unit consuming the value. To synchronize the execution, (j-i) registers are inserted along the bypass. If $i>j$, the register value is generated after it is used if we pipeline the execution without adjustment. Obviously this is not correct. In this case, we need to delay the pipeline execution and add a feedback connection from the function unit which generates the register value to the function unit which consumes it. A total of (i-j) registers are inserted on the feedback connection for synchronizing the execution. An extra (i-j) clock cycle delay is needed besides the normal one clock cycle delay between the pipeline execution of the adjacent iterations. The total delay for this case is (i-j+1), which means that every (i-j+1) clock cycles a new loop iteration begins to execute.

For data dependence from memory operations (load and stores), if the dependence is store after load, we just need to reschedule the store into a pipeline stage that is late enough. If the dependence is store after store, the former store can be eliminated. There is never a dependence between two loads. If the dependence is load after store, the pair of conflicting instructions can be transformed into the instruction that generates the value to be stored, and the instruction which consumes the value load from memory. If this is the case, an extra execution delay or bypass is needed.

For memory data dependences, the dependence could happen beyond adjacent iterations. Since we fetch the iterations consecutively, iteration l and (l+k) have a normal delay of k clock cycles. If the dependent instructions are at the i and j pipeline stages, and the dependence happens between l and l+k iterations, the threshold value for delay or bypass is (j+k-i). If $(j+k-i)>0$, no extra delay is needed. Otherwise, [(i-j-k+1)/k] cycles of extra delay is needed. The delay between adjacent iteration execution is [(i-j+1)/k].

The assumption for pipelining the execution is that function units which take multiple cycles to complete can be internally pipelined. For example, an integer multiplier typically takes 3 cycles. So each multiplier occupies 3 pipeline stages in the datapath. Similarly, memory ports take 3 pipeline stages too.

4.5 Estimation of the Pipeline Execution Time

The pipeline execution of the datapath exploits the maximum parallelism that exists within the loop bodies and between loop iterations. If there is not much parallelism in a loop, it may not be worth switching the execution to the reconfigurable co-processor. If we store the reconfiguration context in a distributed cache on the co-processor to realize fast context switch as proposed, the number of loops to be mapped to the datapath is limited due to context memory size limits. It is important to measure the execution time and speed up we can have by mapping a loop to the datapath. Those loops that can bring maximum speed up should be selected.

Suppose the total number of pipeline stages of a datapath is S, the delay of the consecutive loop iterations is D (≥ 1) cycles, the loop iteration number is N, the execution switch (from general processor to co-processor) overhead is O cycles, and the write back time is W cycles, then the total execution time is given by:

$$T = [S + D*(N-1)] + O + W \quad \text{cycles} \qquad (1)$$

We use this estimation of the execution time for each kernel loop, to select the top kernel loops which can get the most speed up for the application after switching the execution to the reconfigurable co-processor.

5. RECONFIGURABLE DATAPATH

Using the direct mapping methodology and the scheduling algorithm for pipeline execution, we generate the datapath for each selected loop as described in the previous section. The datapath consists of function units and the interconnection between them. Given the multiple datapaths for top kernel loops, we need to generate the common datapath and then reconfigure it into the datapaths for different loops when necessary. The process for merging the multiple datapaths together and generating the common datapath is called datapath mapping. Our previous research [9] has addressed this specific problem in the methodology with a bipartite matching algorithm.

5.1 Datapath Mapping

This part of this methodology has been reported in our prior work and is presented here briefly for review. The datapath mapping algorithm essentially does a graph overlay of the topology graphs for each datapath. The vertices in the topology graph represent function units in the datapath. Vertices have the same color if they represent same type of function units. Directed edges represent the interconnections of function units. The goal is to overlay the graphs, by sharing same colored vertices and interconnections, into a single graph such that it has vertices and interconnections for all the individual graphs and the total number of interconnections is minimized. This combined graph represents the final datapath to be used, with the individual kernel loop datapaths being instantiated using reconfiguration of the interconnect.

5.2 Routing Box

The dotted block in Figure 3 is a function unit in the datapath as viewed from the outside. It consists of three parts. A routing box is placed before the actual function unit and is used to select from

the multiple inputs. A register is inserted after the function unit for pipelining.

In Figure 3, the routing box selects two inputs to the function units from the five possible interconnections coming into this function unit. The more the interconnections, the more the configuration bits needed for this selection. The last element in Figure 3 is the control bits. These are needed during the configuration to specify the function for multi-function units, for example, add vs. subtract. In addition, some constant registers may need to be initialized during the configuration.

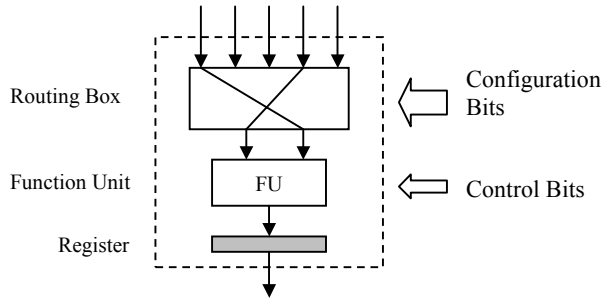

Figure 3. Routing Box for Function Units

5.3 Critical Path and Clock Speed

The critical path in the reconfigurable datapath determines the clock speed in the co-processor. In the reconfigurable datapath, the critical path of a pipeline stage is the critical path of the fixed function unit plus the delay of the routing box plus wire delays. Sometimes a complicated function unit is pipelined inside and executed in multiple clock cycles. Since the functional units are hardwired, their delay is the same as it would be if the functional unit was in the processor. The same is true for the wire delays. As the routing box just needs to select between its inputs, its delay is just one gate delay [9, 10], which is negligible. In general processors, the critical path of pipeline stages is no longer in the functional unit stage, but rather in the more sophisticated branch control and decoding stage. This may be significantly larger than functional unit delay. In fact, two Arithmetic Logic Units (ALUs) on the Pentium 4 processor are clocked at twice the core processor frequency [11]. Thus, it is reasonably safe to assume that the co-processor clock period is less than the master processor clock speed, with the potential for significant speed up.

5.4 Reconfiguration Overhead

Since the reconfigurable datapath is dynamically reconfigured at each context switch with transfer of execution control, the reconfiguration overhead is counted as part of the execution time and is crucial to speed up. The overhead consists of three parts: the reconfiguration context switch, execution switching to the datapath and execution switching back.

Typically a number (usually 8 to 16) of the loops are selected, so we can store the multiple contexts in a distributed cache in the co-processor. During execution, no reconfiguration bits need to be loaded from main memory. We only switch between the multiple contexts, which can be done in 1 clock cycle [2, 9].

Since the trivial part of the codes are executed in the master processor, when we switch the execution to the reconfigurable datapath, live-in registers are read from register files in the general processor to the datapath. After execution, live out registers are written back to register files in the general processor. Usually the 2 sets contain 10~20 registers for each kernel loop. The register file bandwidth in the master processor is the bottleneck for copying the registers. Assuming we have 4 register read/write ports, this results in a 5 cycle overhead.

6. BENCHMARK STUDIES

6.1 MPEG 2 Video Coder

The source code and test data files for this are from Mediabench [4]. The data file for test coding is a bitstream of YUV components (4 frames). The software is compiled and profiled using the IMPACT compiler. The IR we used to extract kernel loop datapaths is generated using the processor platform "EPIC_1G_1BL" without any modern compiler techniques such as superblock, hyperblock or execution predication. "EPIC_1G_1BL" is the EPIC 1-issue 1-branch processor, which we use to represent today's single-issue speculative processor.

The top 10 kernel loops are selected and listed in Table 1 after we do the compilation, profiling, datapath extraction, pipeline scheduling and execution time estimation. These loops constitute 86% of the total execution time.

Table 1. Top 10 kernel loops from MPEG

	Software Time	*Hardware Time*	*# of Memory Ports*	*Speed Up*
Loop 1	820	132.3	2	6.2
Loop 2	85.0	9.57	5	8.88
Loop 3	33.5	4.79	3	6.99
Loop 4	33.1	4.73	3	7.0
Loop 5	35.7	10.8	2	3.31
Loop 6	20.2	0.64	3	31.6
Loop 7	31.4	9.7	2	3.24
Loop 8	11.5	1.82	2	6.32
Loop 9	10.6	1.68	2	6.31
Loop 10	7.3	0.50	2	14.6
Rest	178.2	-	-	1
Overall	1266.3	354.73	-	3.57

(Number in Million cycles)

The software time is the execution time on the general processor in clock cycles. The hardware time accounts for the estimation of pipelined datapath execution time and reconfiguration overhead in clock cycles. Rest stands for the remainder of the code besides the 10 loops, which is always executed in the master processor. Compared with the single-issue processor, we can get a speed up of 3.57. The speed up is based on a conservative assumption of the same clock speed of the co-processor and general processor.

The 10 loops have 7 different datapaths - some of the loops share the same datapath. From the 7 different datapaths, we generate the reconfigurable datapath using the mapping algorithm described earlier. This results in 51 function units and 114 interconnections. Registers are inserted for pipelining and are viewed as part of the function units here. The 51 function units includes 24 registers, 8 adders, 5 memory ports, 3 shifters, 2 multipliers, 2 dividers, 2 logic ALUs, and 5 multiplexers.

An estimate of the total reconfiguration bit size for programming the interconnections is 221 bits. Including further the additional configuration bits for some function units such as adders, shifters and multiplexers, and the configuration bits for initializing some of the constant registers (no more than 8 x 16 bits), each reconfiguration context will be under 500 bits. We can view this as a single instruction for the co-processor needed for the entire loop. To store 5k bits for all 10 loops in the distributed cache in co-processor is very reasonable.

From the datapath mapping result, we see that the hardware resources needed to construct such a reconfigurable datapath are not significant. This makes it feasible to consider multiple copies of this if it can lead to further exploitation of operation level parallelism in the loop. If there is no data dependence between loop iterations, then having multiple copies of the datapath and executing loop iterations in parallel can bring significant speedup. However, this increases the required operand bandwidth, which translates to memory bandwidth. Having multiple copies of the datapath requires duplicating the memory ports too. However, memory bandwidth is likely to be constrained.

It is interesting to examine the relationship between datapath speed up with the memory bandwidth under the assumption of enough other hardware resources in the co-processor. Figure 4 shows the speedup obtained vs. the number of memory ports. The reason for the speed up saturation is the limit on the number of parallel iterations due to inter-iteration data dependency. In this study we have not considered the use of loop restructuring techniques to reduce inter-iteration data dependency. That is likely to further increase the operation parallelism and increase speedup [17]. This is one of the issues we intend to examine in further work.

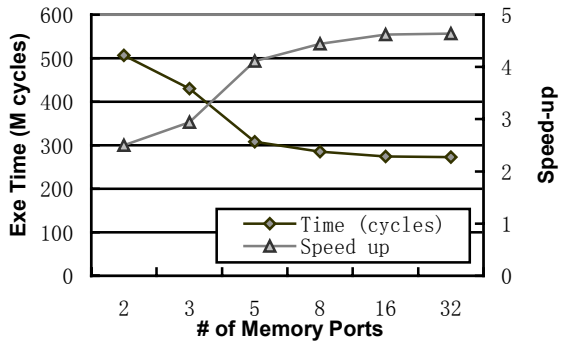

Figure 4. Speed-up vs. Memory Bandwidth for MPEG

Note however that before we see speedup saturation, the number of memory ports is a critical factor that affects the speed up. Usually this number cannot be very large in the current computer systems. Some applications may require a large number of memory ports in order to get a large speed up. We are currently examining techniques for reducing this.

Given that we can view the co-processor as a VLIW with a single instruction per loop, it is instructive to see how this speedup compares with using a conventional VLIW processor. This will provide some insight into the two different ways of harnessing operation level parallelism. The co-processor proposed in this work exploits it through customized datapath synthesis per loop, for the selected loops, and just a regular processor for the rest of the program. In comparison, a VLIW processor exploits it through instruction level parallelism through the entire program. Using the IMPACT compiler, we profiled and estimated the software execution time with different processor platforms. "EPIC_xG_1BL" means x-issue 1-branch EPIC processor. X (1, 4, 8 or 16) instructions can be fetched and issued at the same time. Hyperblocks and execution predication are also used to get the best execution time.

Figure 5 shows the execution times on different EPIC platforms, as well as the execution time of a single-issue processor with a co-processor with 8 memory ports. With 8-issue or 16-issue, a VLIW has almost the same speed for the whole application on the simple-processor, co-processor combination. However the cost of an 8-issue or 16-issue VLIW is large compared to the proposed co-processor. The VLIW processor size grows significantly due to the large number of register file ports, and large instruction decode logic. The program size increases significantly too. Hyperblock formation and predication execution will expand the code size, in this case from 17k instructions to 29k instructions. Large instruction width such as 8 or 16 would bring many empty slots into the VLIW instructions which further expand the code size. The cost of the reconfigurable datapath is much smaller. There is no complicated instruction decode logic, and no large register file ports. The program size can be even smaller since the kernel loop codes become reconfiguration contexts with small bit size. For each kernel loop, the reconfigurable datapath is faster than any EPIC processor. For example, the most executed kernel loop in the MPEG coder takes 128M cycles in an EPIC 8-issue processor while the reconfigurable datapath takes 71.7M cycles with 8 memory ports. By increasing the number of kernel loops mapped onto the co-processor, we can further increase the speedup at the cost of co-processor complexity and program size.

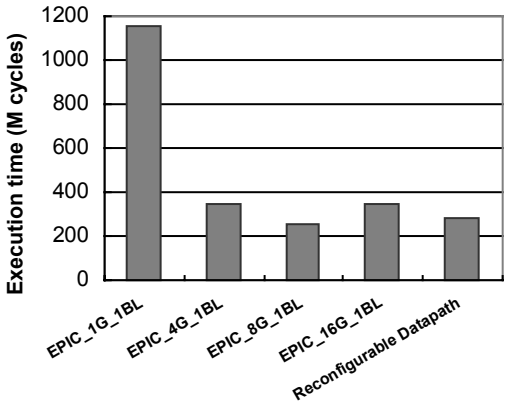

Figure 5. VLIW vs. Reconfigurable Datapath for MPEG

6.2 GSM

GSM (Global System for Mobile communication) is one of the most popular wireless communication standards. We now examine the implementation of a GSM coder, which compresses speech signals. More specifically, GSM 06.10 compresses frames of 160 16-bit linear samples into 33-byte frames. Both the source codes and test data for this are from Mediabench [4].

Table 2 gives the result of speedup of the co-processor against a single-issue speculative processor. An overall speed up of 2.78 for the application is achieved. In the GSM case, the top ten kernel loops account for 81% of the execution time.

The datapath mapping result of GSM shows that the reconfigurable datapath includes 67 function units and 197 interconnections. The 67 function units include 25 registers, 10 memory ports, 17 adders, 8 multipliers, 4 shifters, and 3 multiplexers. An estimate of interconnection reconfiguration size is 415 bits. With additional configuration bits for function units such as adders, shifters, and multiplexers and initialization of constant registers, the total reconfiguration size is expected to be under 1k bits. It is reasonable to store the 10k bits in the distributed cache in the co-processor for the 10 loops.

The speedup obtained vs. the number of memory ports under the assumption of enough other hardware resources so that we can make multiple copies of datapaths is similar to the MPEG case showed in Figure 4. In GSM, with 4 memory ports we could approach the limit of speed up very closely.

The result of comparing the reconfigurable datapath with different EPIC platforms is similar to the MPEG case showed in Figure 5. Again a significantly complex VLIW processor is needed to match the performance of this simple processor, co-processor combination.

Table 2. Top 10 kernel loops from GSM

	Software Time	*Hardware Time*	*# of Memory Ports*	*Speed up*
Loop 1	61.0	9.47	3	6.44
Loop 2	33.3	13.8	2	2.41
Loop 3	5.76	0.203	10	28.4
Loop 4	4.21	0.151	4	27.9
Loop 5	2.31	0.157	1	14.7
Loop 6	2.07	0.192	1	10.8
Loop 7	2.07	0.184	3	11.2
Loop 8	2.06	0.149	2	13.8
Loop 9	1.40	0.118	2	11.9
Loop	1.26	0.149	2	8.46
Rest	26.45	-	-	1
Overall	141.9	51.0	-	2.78

(Number in million cycles)

7. CONCLUSIONS AND FUTURE WORK

This paper describes a methodology for the design of a dynamically reconfigurable datapath co-processor for a specific application (domain). We describe how such a co-processor can be designed and an application mapped to it. Through the two case studies of an MPEG2 encoder and a GSM encoder, we demonstrate how significant speedups compared to a single processor execution can be obtained using relatively little hardware.

We also show how the co-processor can be viewed as a VLIW processor with a single (different) instruction per loop mapped to the co-processor. This then leads to comparing the exploiting of operation level parallelism using such a co-processor with regular VLIW processors. Here we see that significantly larger VLIW processors will be needed to match the performance of such a co-processor, with poorer area and power properties. In general, we can draw some conclusions about the relative applicability of such a co-processor vs. a classic VLIW processor. If the application has a few areas of localized operation level parallelism, such a co-processor provides for a cost (area and power) effective way of harnessing this parallelism without paying the price for it in the rest of the program. However, if the entire application code has operation level parallelism, a VLIW processor is likely to be able to exploit it just as well.

In the above discussion, we have been quite conservative about the parallelism and speed of the co-processor. We have assumed that the co-processor clock speed is the same as the main processor, a fact that we have argued is likely to be conservative. A detailed analysis is needed to determine how much faster the clock on the co-processor can be compared to the main processor. This can lead to further speedup. The VLIW processors we are comparing against have the benefit of the most sophisticated of compilation techniques (hyberblock formation and predicated execution), while our compilation is fairly straightforward. We intend to examine the use of loop restructuring techniques geared towards reducing intra and inter-iteration data dependencies. Further, the function blocks that we consider are in direct correspondence with operations in instructions. We know that many of these can be combined into a single complex function block as is done in application specific processors from Tensilica [15]. Again, this bears further investigation.

Based on the speedups we have seen, and the potential for significant additional speedup based on the directions we have outlined, we believe that these platforms are very interesting for a class of applications with localized sections of operation level concurrency.

8. ACKNOWLEDGEMENTS

This research is supported by the Gigascale Silicon Research Center (GSRC), sponsored by MARCO and DARPA. We would like thanks members of the MESCAL team for interesting discussions as part of this work.

9. REFERENCES

[1] J. R. Hauser and J. Wawrzynek. "Garp: A MIPS Processor with a Reconfigurable Coprocessor," In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM '97, April 16-18, 1997), pp. 24-33.

[2] K. Furuta, T. Fujii, M. Motomura, K. Wakabayashi, M. Yamashina. "Spatial-Temporal Mapping of Real Applications on a Dynamically Reconfigurable Logic Engine (DRLE) LSI," CICC 2000.

[3] IMPACT research group, University of Illinois, at Urbana-Champaign, http://www.crhc.uiuc.edu/IMPACT.

[4] C. Lee, M. Potkonjak, W. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," Micro 30, 1997.

[5] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. Taylor, R. Laufer. "PipeRench: A Coprocessor for Streaming Multimedia Acceleration," *ISCA* 1999.

[6] D. Cronquist, P. Franklin, S. Berg, and C. Ebeling, "Specifying and Compiling Applications for RaPiD," IEEE workshop on FPGAs for Custom Computing Machines, 1998.

[7] M. Wan, H. Zhang, V. George, M. Benes, A. Abnous, V. Prabhu, J. Rabaey "Design Methodology of a Low-Energy Reconfigurable Single-Chip DSP System," Journal of VLSI Signal Processing, 2000.

[8] Chameleon Systems Inc., San Jose, CA, "CS2000 Reconfigurable Communications Processor Product BRIEF" from http://www.chameleonsystems.com.

[9] Z. Huang and S. Malik, "Managing dynamic reconfiguration overhead in system-on-a-chip design using reconfigurable datapaths and optimized interconnection networks", in DATE 2001.

[10] Product Data Sheets, "Virtex: DC and Switching Characteristics", from http://www.xilinx.com.

[11] Intel Data Sheet, "Product overview: Intel® Pentium® 4 processor-based workstations," from http://www.intel.com.

[12] "System Level Design: Orthogonolization of Concerns and Platform-Based Design", K. Keutzer, S. Malik, J. M. Rabaey, A. R. Newton and A. Sangiovanni-Vincentelli), IEEE Transactions on Computer-Aided Design, Vol. 19, No. 12, December 2000.

[13] T. J. Callahan and J. Wawrzynek, "Instruction-Level Parallelism for Reconfigurable Computing", *Field-Programmable Logic and Applications FPL'98*, Tallinn, Estonia, Aug-Sep 1998.

[14] TriMedia Technologies Inc., http://www.trimedia.com/.

[15] Tensilica Inc., http://www.tensilica.com/.

[16] Morphics Technology Inc., http://www.morphics.com/.

[17] M. Wolfe, "High Performance Compilers For Parallel Computing", Addison-Wesley Publishing Company, 1996.