Kluwer Journal on Design Automation of Embedded Systems, November 2002, Vol. 7, No. 4, pp.325 -339.

Improving Software Performance with Configurable Logic

Jason Villarreal, Dinesh Suresh, Greg Stitt, Frank Vahid, Walid Najjar
Department of Computer Science and Engineering
University of California, Riverside
{villarre, dinesh, gstitt, vahid, najjar}@cs.ucr.edu

Abstract. We examine the energy and performance benefits that can be obtained by re-mapping frequently executed loops from a microprocessor to reconfigurable logic. We present a design flow that finds critical software loops automatically and manually re-implements these in configurable logic by implementing them in SA-C, a C language variation supporting a dataflow computation model and designed to specify and map DSP applications onto reconfigurable logic. We apply this design flow on several examples from the MediaBench benchmark suite and report the energy and performance improvements.

1. Introduction

Microprocessors are increasingly being supplemented with configurable logic, such as field-programmable gate arrays (FPGAs), in embedded systems. A typical product may include numerous peripherals or custom logic components previously implemented as an application-specific integrated circuit (ASIC), in addition to a microprocessor IC. Due to the time and cost of designing and building ASICs, designers are increasingly implementing those additional peripherals and components in configurable logic. Furthermore, recent low-cost, mass-produced devices incorporating both a microprocessor and configurable-logic make such implementations even more attractive.

Given the increasing appearance of configurable logic along with microprocessors in embedded systems, we set out to develop a method for using a small amount of that logic to improve the performance of the microprocessor software. Our method utilizes three steps: software loop analysis, critical-loop recoding and compilation, and standard synthesis. Our method fits into existing design flows and requires a relatively short amount of time to apply to a given application.

To support this method, we developed a general tool for analyzing loops. The tool, called LOOAN, can be integrated with a variety of different processor simulators - we have so far integrated it with SimpleScalar [4] , an 8051 simulator [22], and a MIPS simulator [6], all of which are publicly available.

For our critical-loop recoding and compilation, we utilize SA-C [11] - Single-Assignment C, and its associated compiler and VHDL generation tool. SA-C was originally developed for capturing image processing algorithms and supports a dataflow computation model. While similar to C, the combination of SA-C's restrictions along with its additional constructs enable C programmers to straightforwardly describe such algorithms and perform extensive optimizations that otherwise would have been difficult to apply on regular C code. In our method, we recode the most critical loops, as determined from our loop analysis, in SA-C, and apply the SA-C compiler, which generates structural VHDL code.

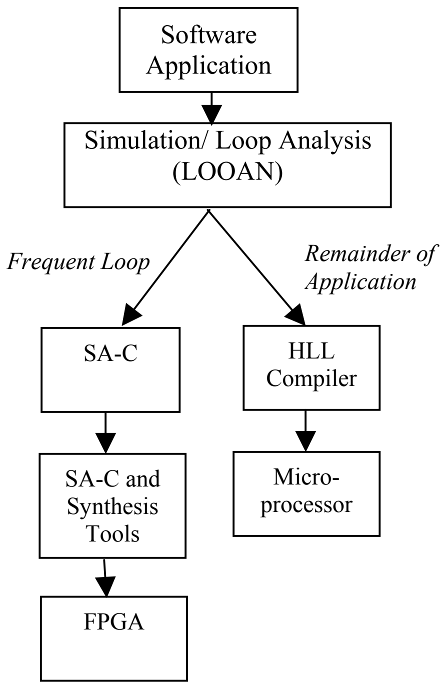
The last step of synthesis can be carried out by any of a variety of commercial synthesis tools.

In this paper, we highlight the LOOAN [23] loop analysis and SA-C compiler tools, and we describe how our method can be applied by an embedded system designer wishing to improve software performance by using configurable logic. To validate the method, we applied it to several MediaBench benchmarks [17]. We show significant software speedups can be obtained by using just a moderate amount of configurable logic. In addition, we achieve significant energy improvements, which is important for battery-operated embedded systems.

2. Previous Work

Partitioning an application among software and hardware has been investigated from several different viewpoints in the past. Several partitioning tools have focused on automated partitioning of a program among a microprocessor and a custom processor (implemented on an ASIC or configurable logic) [13][14][5][9][15]. These approaches read an executable specification into an internal representation, possibly annotated with frequencies determined through profiling, and then

Figure 1: Hardware/software partitioning design flow.



apply automated partitioning heuristics of that representation. Such partitioners represent sophisticated tools that can be challenging to incorporate into an existing tool flow.

Much work has been done on augmenting microprocessors with FPGAs. PRISM [1] is a high-level language approach which partitions high level code onto a board consisting of a Motorola 68010 and four Xilinx 3060 FPGAs. Napa C [7] takes a similar approach by compiling C code to hybrid RISC/FPGA architectures. The Garp architecture [12] combines a MIPS microprocessor core with reconfigurable logic.

Recent efforts have focused on generating custom instructions to improve software performance. Tensilica [8] and HP's PICO [19] focus on synthesis of customized processors. Kucukcakar [16] considers using configurable logic in the datapath to implement custom instructions.

In contrast to the above approaches, our method does not impose a heavy impact on the tool flow. We instead extracted the key element of automated partitioners - the detection of the most critical loops - and made it a separate tool that could be easily interfaced with different instruction-set simulators. Our method uses a simple microprocessor/configurable-logic architecture without requiring special methods of integrating the two - they merely need to share memory. Our method does not change the microprocessor instruction set, but can be complementary to approaches that do.

3. Design Flow

With microprocessor/reconfigurable logic based systems becoming more common, it becomes increasingly important to use the reconfigurable logic efficiently. One such way to use this reconfigurable logic is to implement only the most-critical regions of a software application on this logic in order to improve the performance of the application. It is a general rule that most of the time spent in software is spent within a small portion of the code. However, based on the application, it is generally difficult to identify at compile time where this small portion of the code lies. In order to identify the critical section of the code, an embedded system designer needs to perform profiling and loop analysis of the high level code (usually C) as a first step towards optimizing.

Once loop analysis has been done, the designer can identify the loops or functions that are the most time-dominating and then partition them onto hardware. Several choices exist on how to implement these loops in hardware, with various advantages and disadvantages to each approach. It would be beneficial if the loops that were being taken to hardware could be implemented in a high level language. Following the partitioning, the designer should be able to simulate the result or run it on actual hardware and observe the energy and performance savings.

This design flow is illustrated in Figure 1. The design flow starts from a high-level software description of the application. This application is then executed on a simulator that provides data for the loop analysis. From the results of the loop analysis, the most frequent loop is determined and then partitioned into hardware while the rest of the application is still implemented in software. The loop is then re-implemented into SA-C code. The SA-C tools can then generate a hardware description, which can be synthesized by standard synthesis tools and placed in an FPGA. The remaining software is compiled with any typical high-level language compiler and run on a microprocessor.

3.1. LOOAN

In order to profile benchmarks, we introduce an automatic tool set that performs loop analysis and profiling. The tool reads a benchmark's assembly file, map file (which contains the locations of functions in memory), and instruction trace and creates a directed acyclic graph (DAG) representation in which the root of the DAG has children that correspond to all of the routines in the code, e.g., main, printf, etc. Each routine node has children nodes that correspond to that routine's loops, which are automatically numbered beginning with 1. Similarly, each loop node has children nodes that correspond to that loop's sub-loops. Routine calls are also linked in through a special node linked as a child of the loop in which it appears.

After the DAG is created, the loop analysis program will parse the instruction trace and update each node with usage information. After we have processed the entire instruction trace, we calculate certain statistical data and output the information to a file including dynamic instruction counts per loop, average number of iterations per loop, and number of times each loop is called.

Collectively, we refer to this set of tools as LOOAN [23] (which stands for LOOp ANalysis). The LOOAN tool set supports applications compiled for the MIPS, the 8051, and was extended to support SimpleScalar for this paper. The tool itself was written in standard C++ and can be executed on a variety of platforms.

We chose the above approach over a binary instrumentation approach for several reasons. One was that we could easily update our analysis program to keep additional statistics. A second is because the above approach yields no change in program behavior. The disadvantages compared to instrumentation are the slower execution and the need to generate large trace files.

3.2. SA-C

In order to implement the critical sections located by LOOAN into hardware, the designer needs to implement them in a language that can be translated to hardware. One way to accomplish this is to code the loops directly in a hardware description language like VHDL, but this can have some drawbacks. First, software writers generally are not familiar with hardware description languages. Second, the common tools that synthesize from hardware description languages generally operate at the register-transfer level – behavioral level tools are not common. Another approach is to use a high-level language, but most high-level languages are not designed to go to hardware. SA-C, however, is one such high-level language that is specifically designed to generate hardware.

SA-C, which stands for Single Assignment C, is a variant of C and has been designed to express Image Processing (IP) applications at a high level, while being amenable to efficient compilation to fine grain parallel hardware systems[2][3]. As the name suggests, SA-C's most important restriction in comparison to C is that the value of any variable can be set only once, when the variable is declared. This single assignment restriction is found in many functional programming languages, and has the property that it breaks the von Neumann equivalence between variables and memory locations. Since variables can be set only once, they correspond to values (not addresses) and can be assigned directly to wires. SA-C also removes the C de-referencing and address operators (* and &), thus eliminating pointers, and forbids recursion.

One of the main advantages of SA-C is that it hides the details and intricacies of low-level hardware design from the application programmer. At the same time, the SA-C compiler leverages extensive optimizations and code transformations to increase the speed and reduce the size of the resulting circuit.

SA-C programs are compiled to FPGA configurations plus a C program that manages the FPGA in terms of downloading the configuration and data, triggering the FPGA, and uploading the results. Thus, from the point of view of an application developer, SA-C programs are like any program running on a more traditional processor. The compiler maps SA-C programs to executables, which are invoked like any other program on the host. The only indication that part of the program was actually mapped to a circuit and executed on a reconfigurable co-processor is its speed of execution.

Figure 2: Comparison of C code (a) and SA-C code (b).

a)

```
for (im_pos=x_pos+y_im_lin;
     x_filt<y_filt_lin;
     x_filt++,im_pos++)
        sum+=image[im_pos]*temp[x_filt];
```

b)

```
int32 main(int16 image[:],
           int16 temp[:],
           int16 im_pos, int16 x_filt,
           int16 iterations)
  {
    uint32 res =
        for i in
         image[im_pos : im_pos + iterations]
        dot
         temp[x_filt:x_filt + iterations]
             return(sum((uint32)i * j));
  }return (res);
```

The SA-C compiler supports a wide range of optimizations aimed at producing a more efficient hardware execution. Most of these optimizations are aimed at reducing the size of the circuit (e.g., common sub expression elimination), reducing the propagation delay of the circuit (e.g. pipelining), or reducing the I/O requirements of the circuit (e.g. strip-mining, which reorders loops to better access the memory hierarchy). SA-C optimizations also include traditional optimizations such as constant folding, operator strength reduction, function in-lining, dead code elimination, invariant code motion and common sub-expression elimination. Other optimizations are specific to SA-C or are adapted from vector and parallel compilers or synthesis tools[10].

One important aspect of SA-C is that it was never conceived to be a stand-alone language, rather it is assumed that selected loops and functions of existing C programs would be re-written in SA-C and incorporated in the original program. The SA-C compiler would then map these segments to hardware. In fact, SA-C does not support any file I/O operation, it is assumed that such operations are carried out in C.

Figure 2(a) shows C code taken from the innermost loop of the internal_filter function in the epic benchmark. The loop operates on two arrays and returns the sum of the dot product over a section of these arrays. The arrays *image* and *temp* are indexed by the variables *im_pos* and *x_filt* respectively.

Figure 2(b) shows the equivalent SA-C code. In order to run this loop in hardware, it needs to be written as a standalone function. The input arrays, initial values of the array indices, and the number of iterations are passed as parameters to the SA-C function. The parameter *iterations* corresponds to the number of loop iterations in the C code and in this example is equal to *y_filt_lin* – the initial value of *x_filt*. Here the choice of the input data type is dependent on the nature of the input. To execute this benchmark on hardware we generated 16 inputs, and hence have declared the arrays to hold 16-bit integers.

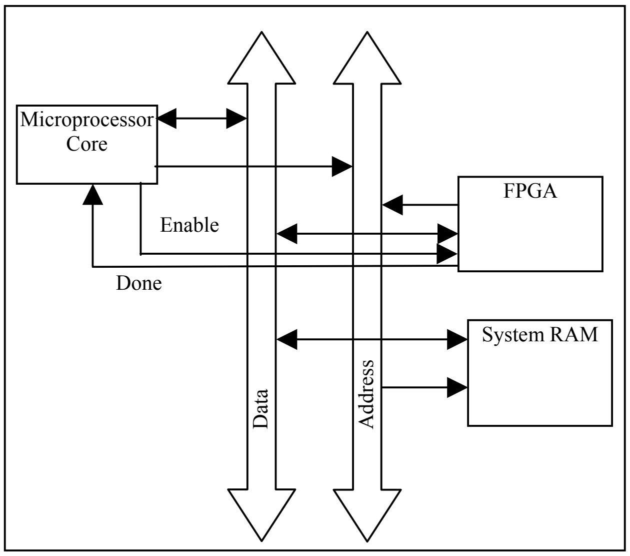
In SA-C, any block of statements must return a value. In Figure 2(b), the for loop computes the dot product of the arrays in the given index range. The sum operator computes the sum of all the values generated during each iteration. Note that *dot* and *sum* are built-in operators in the SA-C language. The sum of the dot products is stored in the variable *res* and is returned as the final return value in the program. Note that this is functionally equivalent to the C code.

3.3. Target Architecture

One of the many target architectures that could be applied to this design flow is illustrated in Figure 3. This architecture consists of a microprocessor and an FPGA connected over a memory bus. In addition to sharing memory, the processor has direct control over the activation of the FPGA through an *enable* signal. Similarly, the FPGA has a *done* signal that can communicate directly to the processor and signal completion. If an application was targeted towards an architecture of this type, then the microprocessor would execute software instructions until it jumped to the portion of code that was partitioned onto the FPGA. The microprocessor would wait in a low-power state until the FPGA had finished executing and signals it was done. We chose this approach because we did not try to extract any parallelism from the code but merely speed up a portion of it. Future work includes looking at running the FPGA and processor in parallel to achieve even greater performance benefits.

This architecture is a simplified version of architectures that are commercially available today, such as Triscend's E5 and A7 chip [21]. The E5 combines an 8051 microprocessor with configurable logic. The A7 is similar except it uses an ARM processor. Both of these systems have components called configuration registers, which are writable in software, but can be read by the custom logic. Our target architecture assumes that enabling the custom logic uses a configuration register.

Figure 3: Target Architecture



There is an additional component on the Triscend chips, called a status register, which is only writable by the custom logic but can be read by the software. We assume that a status register is used to specify the end of execution in the custom logic.

The Triscend architectures are slightly more complex than our target architecture and generally contain components such as a DMA in order to share the memory. Our architecture, however, does not require a DMA because we are guaranteed that the time the processor executes and the time the FPGA executes will be mutually exclusive. The FPGA and processor both access memory by writing to the address bus and reading from the data bus. The FPGA modifies any variable or memory location it needs to as normal and the processor will be able to see these changes when it restarts. There are also no peripherals on our simplified architecture that would interrupt the processor or need access to the memory. There is currently no cache in this architecture, as this is left as future work.

4. Experiments

4.1. Loop Analysis

For our experiments, we use the MediaBench [17] benchmarks. MediaBench is a benchmark suite of multimedia applications, specifically designed for embedded systems. We followed the design flow described in Section 3 on five of the MediaBench benchmarks: EPIC encoding (Efficient Pyramid Image Coder), G721 decoding (voice decompression), MPEG decoding (video decompression), Pegwit (public-key generation), and JPEG Decoding (still image decompression). We compiled these benchmarks for SimpleScalar and simulated them using *sim-outorder*, the out-of-order execution SimpleScalar simulator. We simulated each benchmark on the data set provided by the MediaBench distribution. By using the built-in ptrace flag of *sim-outorder*, we generated a trace file for each benchmark. This, along with an assembly file generated by one of the binary utilities provided by SimpleScalar, was given as input to LOOAN.

The loop structure and usage results from LOOAN are reported in Table 1. In the table, the topmost entry of each benchmark is the statistic for the entire program. Below that are the functions and loops that contribute more than 15% to the total dynamic instruction count of the program, decreasingly ordered based on their contribution to the total dynamic instruction count (labeled TDI in the table). Loops are numbered based upon the static order they appear in the assembly file. Thus, the loop <internal_filter>.3 represents the third loop located in the internal_filter function. Subloops are similarly numbered, so <internal_filter>.3.1 represents the first subloop located inside the third loop of the internal_filter function. The entire function itself is represented by the name without a number following it. The table reports the number of executions of each loop, which is defined as the case when a loop is entered from code outside the loop, as well as the number of iterations per execution, where an iteration is defined to be a pass through the body of the loop followed by a jump to the beginning of the loop.

Table 1: Loop analysis results.

Loop	Size	Instr/Exec.				Iter/Exec.				Execs	TDI	% of
	(Instrs)	avg	min	max	stddev	avg	min	max	stddev			Program
EPIC Encoding												
.	154009	1586547.54	2	49963774	6518079.28	1	1	1	0	72	114231423	100.00%
..<internal_filter>	7049	145088.9	219	11374866	1092008.2	1	1	1	0	672	97496885	85.35%
..<internal_filter>.3	617	3639304.38	84550	11374840	4550482.25	75	9	255	68.18	24	87343305	76.46%
..<internal_filter>.3.1	473	48507.16	13	94791	31012.86	87.79	1	128	42.14	1800	87312896	76.44%
..<internal_filter>.3.1.1	281	378.59	3	729	290.48	6.68	1	16	6.92	218984	82905440	72.58%
..<internal_filter>.3.1.1.1	153	44.79	2	301	62.44	2.67	1	16	3.26	1401460	62766187	54.95%
G721 Decoding												
.	95017	1674.86	11	268566	4255.4	1	1	1	0	320311	536477171	100.00%
..<quan>	241	192.59	4	249	50.22	1	1	1	0	1333098	256739532	47.86%
..<quan>.1	137	181.82	1	236	45.82	11.67	1	15	2.79	1318474	239724588	44.68%
..<update>	4745	1153.88	17	1438	178.02	1	1	1	0	131469	113595109	21.17%
MPEG Decoding												
.	197321	10238	11	8674768	90501.44	1	1	1	0	42580	435933911	100.00%
..<Reference_IDCT>	1385	5273.39	79	46795	11608.8	1	1	1	0	69185	350464223	80.39%
..<Reference_IDCT>.2	689	2771.07	74	24936	5635.37	1.92	1	9	1.95	69185	177340943	40.68%
..<Reference_IDCT>.2.1	601	1441.29	16	3167	1141.05	4.82	1	9	3.11	132545	176659823	40.52%
..<Reference_IDCT>.1	585	21846	21846	21846	0	9	9	9	0	7920	173020320	39.69%
..<Reference_IDCT>.1.1	497	2417.78	16	2718	849.16	8.11	1	9	2.51	71280	172339200	39.53%
..<Reference_IDCT>.1.1.1	305	277.19	11	314	98.16	8.01	1	9	2.63	578160	160261200	36.76%
..<Reference_IDCT>.2.1.1	297	271.93	11	308	96.18	8.01	1	9	2.63	578160	157219920	36.07%
Pegwit												
.	199913	126146.35	10	2170599	107468.99	1	1	1	0	503	63451614	100.00%
..<gfAddMul>	1305	381.55	2	1260	374.25	1	1	1	0	71947	27451280	43.26%
..<gfMultiply>	1833	19687.73	18826	21117	281.52	1	1	1	0	1298	23985456	37.80%
..<gfAddMul>.2	489	374.01	9	1188	343.89	7.95	1	27	6.68	62297	23300003	36.72%
..<gfMultiply>.2	713	17930.8	17069	19360	281.31	18	18	18	0	1298	23274175	36.68%
..<gfMultiply>.2.1	489	1026.52	892	1171	48.79	18.43	17	24	0.7	22066	22651134	35.70%
JPEG Decoding												
.	355065	4441.83	5	347756	24054.33	1	1	1	0	1885	8372848	100.00%
..<jpeg_idct_islow>	7817	4805.36	1732	6799	1150.93	1	1	1	0	851	4089358	48.84%
..<jpeg_idct_islow>.2	3777	3084.07	862	3413	809.28	9	9	9	0	851	2624540	31.35%
..<ycc_rgb_convert>	1017	11914	11914	11914	0	1	1	1	0	149	1775186	21.20%
..<ycc_rgb_convert>.1	769	11883	11883	11883	0	2	2	2	0	149	1770567	21.15%
..<ycc_rgb_convert>.1.1	417	11817	11817	11817	0	228	228	228	0	149	1760733	21.03%
..<jpeg_idct_islow>.1	3817	1694.29	843	3393	534.27	9	9	9	0	851	1441841	17.22%

As evidenced by this table, only a very small portion of the code contributes to the majority of the execution time.

The loops we chose to implement were <internal_filter>.3.1.1 for EPIC, <quan>.1 for G721, <Reference_IDCT>.1 for MPEG, <gfAddMul>.2 for Pegwit, and <jpeg_idct_islow> for JPEG. We chose loops that were frequent but would fit within an FPGA and would not be overly difficult to code. For MPEG, the loops <Reference_IDCT>.2 and <Reference_IDCT>.1 take approximately the same percentage of time but <Reference_IDCT>.2 included a function call. Implementing functions in hardware requires them to be inlined, which we plan to investigate as future work.

4.2. SA-C

Once we had identified the loops that took up the majority of the execution time, we coded them up in SA-C. This process took between one and two hours per benchmark. Several of the benchmarks used floating point numbers in their implementations. Floating point operations are not efficiently implemented on FPGAs, so when coding them we converted them to a fixed point implementation that had similar functionality. The conversion from a floating point to a fixed point constant size was dependent on the precision we needed for the application. For example, a fixed-point representation of 21 bits for the integral portion and 10 bits for the decimal portion provided reasonably accurate results for the JPEG benchmark.

Converting the critical loops from C to SA-C was a straightforward process, but not automated. Because SA-C is a single assignment language, it is not possible to directly code up a loop that reuses the same memory location. Instead, one must discover the access pattern inherent in the loop, be it dot product of vectors, cross product of vectors, etc., and apply it to generate the correct results. Certain C conventions make it difficult to convert to SA-C, such as the use of incrementing pointers to go through arrays and any recursive functions that are difficult to represent iteratively. Currently, manual inspection of the code is required to detect these patterns.

Once a benchmark has been coded up in SA-C, in a single step we can get it running on hardware as well as get information on the area it needs (number of configurable logic blocks, number of slices, number of flip-flops, number of look-up tables and estimated clock speed). With the information generated, we can perform power estimation.

4.3. Performance and Energy Savings

We used a simulation-based approach for performance evaluation. For the microprocessor core shown in Figure 3, we use a 32-bit MIPS. The number of instructions executed during each application and loop are obtained from the loop analysis tool. We calculate total cycles for software execution based on a cycles per instruction count (CPI) of 1.5, which was estimated based on typical results from a MIPS simulator [6]. Given the clock frequency, we can determine the total time taken by the loop and the entire application. The execution time of all the software excluding the loop is determined by subtracting the loop time from the total time. We estimate the execution time of the FPGA in a similar manner. We first determine the number of cycles required by each iteration of the loop. Since the performance of the hardware is limited by one memory access per cycle, the number of cycles per iteration in the loop is generally the number of memory accesses. The hardware generated by SA-C is usually able to reach the one memory access per cycle limit by performing optimizations such as pipelining, loop unrolling, etc. After determining the number of cycles per iteration, we can determine the total number of cycles required by the hardware by using the number of iterations per execution of the loop and the total amount of executions of the loop. We obtain the clock frequency for the FPGA after the design has been placed and routed, and use this frequency to determine total execution time of the hardware. Frequencies for our designs typically ranged from 40 MHz to 50 MHz. Total execution time after implementing the most frequent loop in hardware is determined by adding the execution time of the hardware with the time of everything but the loop in software.

Power and clock frequency of the MIPS were obtained from the MIPS32 4KP core on the MIPS website [18]. It is quite common for architectures that combine a microprocessor and configurable logic to run the microprocessor at a much slower frequency than would typically be used. In order to estimate results for these types of systems, we use two different clock frequencies. In one set of experiments, we use a frequency of 100MHz for the MIPS in order to represent systems with a reduced clock. We also used a clock frequency of 200MHz, which is reported as a typical frequency for the MIPS32 4KP [18], in order to represent higher performance systems.

Table 2: Microprocessor/FPGA performance and energy improvements.

	Performance (kilo-cycles)						100 MHz MIPS Results			200 MHz MIPS Results		
Eg	Sw	Loop in sw	Loop in FPGA	Sw / FPGA	FPGA Clock (MHz)	Potential Speedup	Power (W)	Energy Savings	Speed up	Power (W)	Energy Savings	Speed up
g721	804,716	359,387	22,414	467,743	45	1.8	0.39	32%	1.6	0.47	24%	1.5
jpeg	12,559	6,134	21	6,447	36	2.0	0.38	44%	1.9	0.45	44%	1.9
pegwit	95,177	34,950	1,994	62,221	50	1.6	0.40	23%	1.5	0.48	18%	1.4
epic	171,347	94,149	14,491	91,689	42	2.2	0.46	38%	2.1	0.56	12%	1.5
mpeg	635,901	259,530	4,625	380,996	30	1.7	0.39	30%	1.6	0.47	28%	1.5
						Avg:	**0.40**	**33%**	**1.8**	**0.48**	**25%**	**1.6**

All hardware generated from SA-C is synthesized using Synplify [20]. All mapping, placing and routing is handled by Xilinx tools. We used Xilinx's Virtex Power Estimator [24] to estimate FPGA power, utilizing a 0.18 micron FPGA technology (in particular, the XCV2000E). All power analysis was done using 0.18 micron technology running at 1.8 V.

In order to determine total power of the system, we measured actual devices that are similar to our architecture. The Triscend E5 [21] is one such device, which combines an 8051 microprocessor with configurable logic. We determined that the microprocessor low-power state on the Triscend E5 was 85% of its active state, and low-power state of the configurable hardware was 12.5% of its active state, and thus used the following equation to compute total power:

$$\text{Total power} = \%Sw * (Sw + .125*Hw) + \%Hw * (Hw + .85*Sw)$$

where %Sw is the percent of time spent in software, %Hw the percent time spent in the FPGA, Sw is the power of the software when the microprocessor is active, and Hw is the power of the FPGA when active.

Power and performance results for the MIPS architecture are shown in Table 2. *Sw* is the performance of the application when running completely in software, which is determined by multiplying the number of total dynamic instructions reported by LOOAN by 1.5, which is the approximate cycles per instruction of our MIPS simulator [6]. *Loop in sw* is the performance of the loop when running in software, which is the total number of dynamic instructions of the loop reported by LOOAN multiplied by 1.5. *Loop in FPGA* is the performance of the loop when implemented in hardware running on the FPGA, this was calculated by an analysis of the VHDL code to determine the latency of each execution from memory accesses and computational delays. All performance results are reported in kilo-cycles. *FPGA clock* is the clock frequency for the loop when running on the FPGA. *Potential speedup* is the maximum possible speedup, according to Amdahl's Law, that can be achieved by partitioning the loop into hardware assuming the loop takes zero time. *Power* is the overall power of the system in Watts. *Energy savings* and *speedup* represent the improvements made after implementing the loop in the FPGA.

Notice that two sets of results are given, one for a system using a 100 MHz microprocessor and another using a 200 MHz processor. Significant speedups are achieved for both sets of results. For the 100 MHz-microprocessor system, we achieved an average speedup of 1.8. For the 200 MHz system, the average speedups were reduced to 1.6. The large difference between these results occurs because at 200 MHz, the processor's clock is between 4 and 5 times faster than the clock on the FPGA, which implies that performance improvements of the hardware become less significant. Notice that all speedups are somewhat close to the potential speedups determined by Amdahl's Law.

An important point to notice is that energy savings tend to be less than the performance improvements. This occurs because of an increase in total power, due to the high power consumption of the FPGA. In general, an FPGA typically consumes much more power than a microprocessor. In our experiments, the power of the microprocessor ranged from .07 Watts to .140 Watts. However, the power of the FPGA ranged from .123 to .233 Watts, excluding quiescent power of .270 Watts. Therefore, the FPGA tends to consume at least twice as much power, even at a lower clock frequency. Average energy savings for the 100 MHz system are 33%. For the 200MHz system, the average energy savings are 25%. The energy savings on the 200 MHz MIPS for JPEG can be explained by the fact that SA-C was designed for image processing applications and could optimize the hardware to a greater degree than the other benchmarks.

5. Conclusions

In this paper, we study the potential performance and energy benefits from partitioning a frequent loop of an application onto configurable hardware. We show that using a design flow consisting of loop analysis, critical loop recoding into SA-C and compilation, and standard synthesis, is sufficient for significant improvements. We achieve speedups ranging from 1.5 to 2.1 on systems with clocks running at 100 MHz and 200 MHz. In all examples, we achieve the added benefit of energy savings, generally ranging from 12% to 44%. Future work includes inlining functions in loops, looking at different architectures, and automating the recoding of frequent loops.

6. Acknowledgements

This research was supported in part by NSF ITR Award 0083080, DARPA AFRL contract F33615-98-C-1319 , and by a grant from Trimedia Technologies, Inc.

References

[1] P. Athanas, H. Silverman. Processor reconfiguration through instruction-set metamorphosis. Computer, Volume: 26 Issue: 3 , March 1993 Page(s): 11 –18.

[2] W. Bohm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, and W. Najjar. Mapping a Single Assignment Programming Language to Reconfigurable Systems. Supercomputing, 21:117-130, 2002.

[3] W. Bohm, B. Draper, W. Najjar, J. Hammes, R. Rinker, M. Chawathe and C. Ross. One-Step Compilation of Image Processing Applications to FPGAs. IEEE Symposium on Field-programmable Custom Computing Machines, Rohnert Park, CA, April 30 - May 2, 2001.

[4] D. Burger, T. Austin and S. Bennett. "Evaluating Future Microprocessors: The SimpleScalar ToolSet". University of Wisconsin-Madison. Computer Science Department. Technical Report CS-TR-1308, July 1996.

[5] P. Eles, Z. Peng, K. Kuchchinski and A. Doboli. System Level Hardware/Softeare Partitioning Based on Simulated Annealing and Tabu Search. Kluwer's Design Automation for Embedded Systems, vol2, no 1, pp. 5-32, Jan 1997.

[6] T. Givargis, F. Vahid, and J. Henkel. System-Level Exploration for Pareto-Optimal Configurations in Parameterized Systems-on-a-Chip. International Conference on Computer-Aided Design (ICCAD), San Jose, November 2001.

[7] M. Gokhale, J. Stone. NAPA C: Compiling for hybrid RISC/FPGA architectures. IEEE Symposium on FPGAs for Custom Computing Machines, FCCM '98.

[8] R. Gonzalez, R.E. Xtensa: A Configurable and Extensible Processor. IEEE Micro, pp. 60-70, 2000.

[9] J. Grode, P. Knudsen, J. Madsen. "Hardware Resource Allocation for Hardware/Software Partitioning in the LYCOS System." Proc. of the 1998 Design Automation and Test in Europe.

[10] J. Hammes, W. Bohm, C. Ross, M. Chawathe, B. Draper, R. Rinker, and W. Najjar. Loop Fusion and Temporal Common Subexpression Elimination in Window-based Loops. IPDPS 8th Reconfigurable Architectures Workshop, San Francisco, CA, April 27, 2001.

[11] J. Hammes, R. Rinker, W. Najjar, B. Draper. A High-level, Algorithmic Programming Language and Compiler for Reconfigurable Systems. The 2nd International Workshop on the Engineering of Reconfigurable Hardware/Software Objects (ENREGLE), part of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), Las Vegas, NV, June 26-29, 2000.

[12] J. Hauser, J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. IEEE Symposium on FPGAs for Custom Computing Machines, pages 12-21, Napa Valley, CA, April 1997.

[13] J. Henkel, Y. Li. Energy-conscious HW/SW-partitioning of embedded systems: A Case Study on an MPEG-2 Encoder. Proceedings of Sixth International Workshop on Hardware/Software Codesign, March 1998, pp. 23-27.

[14] J. Henkel. A low power hardware/software partitioning approach for core-based embedded systems. Proceedings of the 36th ACM/IEEE conference on Design automation conference, pp. 122 – 127,1999.

[15] A. Kalavade and E.A. Lee. The Extended Partitioning Problem: Hardware/Software Mapping, Scheduling and Implementation-Bin Selection. Kluwer Design Automation for Embedded Systems, vol 2, no 2, pp. 125-163, Mar 1997.

[16] K. Kucukcakar. An ASIP Design Methodology for Embedded Systems. International Symposium on Hardware/Software Codesign, May 1999.

[17] Lee, C., M. Potkonjak, and W.H Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," Proc. 30th Annual International Symposium on Microarchitecture, Dec. 1997, pp. 330-335.

[18] MIPS Technologies, Inc. http://www.mips.com.

[19] R. Schreiber et al., "High-level synthesis of nonprogrammable hardware accelerators." Proceedings of the International Conference on Application-Specific Systems, Architectures, and Processors. pp 113-124, July 2000.

[20] Synplicity, www.synplicity.com/products/synplifypro.html.

[21] Triscend Corporation, http://www.triscend.com.

[22] UCR Dalton Project. http://www.cs.ucr.edu/~dalton/

[23] J. Villarreal, R. Lysecky, S. Cotterell, and F. Vahid. Loop Analysis of Embedded Applications. UC Riverside Technical Report UCR-CSE-01-03, 2001.

[24] Virtex Power Estimator, http://support.xilinx.com/cgi-bin/powerweb.pl.