Energy Savings and Speedups from Partitioning Critical Software Loops to Hardware in Embedded Systems

GREG STITT, FRANK VAHID, and SHAWN NEMATBAKHSH
University of California, Riverside

We present results of extensive hardware/software partitioning experiments on numerous benchmarks. We describe our loop-oriented partitioning methodology for moving critical code from hardware to software. Our benchmarks included programs from PowerStone, MediaBench, and NetBench. Our experiments included estimated results for partitioning using an 8051 8-bit microcontroller or a 32-bit MIPS microprocessor for the software, and using on-chip configurable logic or custom application-specific integrated circuit hardware for the hardware. Additional experiments involved actual measurements taken from several physical implementations of hardware/software partitionings on real single-chip microprocessor/configurable-logic devices. We also estimated results assuming voltage scalable processors. We provide performance, energy, and size data for all of the experiments. We found that the benchmarks spent an average of 80% of their execution time in only 3% of their code, amounting to only about 200 bytes of critical code. For various experiments, we found that moving critical code to hardware resulted in average speedups of 3 to 5 and average energy savings of 35% to 70%, with average hardware requirements of only 5000 to 10,000 gates. To our knowledge, these experiments represent the most comprehensive hardware/software partitioning study published to date.

Categories and Subject Descriptors: C.3 [**Special Purpose and Application-Based Systems**]: Real-Time and Embedded Systems

General Terms: Design, Performance

Additional Key Words and Phrases: Hardware/software partitioning, FPGA, synthesis, platforms, low energy, speedup, embedded systems

1. INTRODUCTION

Much previous work has shown the advantages of hardware/software partitioning in embedded system design. Hardware/software partitioning divides an application into software running on a microprocessor and some number of coprocessors implemented in custom hardware. Advantages of such partitioning include improvement in performance (e.g., Gokhale and Stone 1998; Hauser

This research was supported in part by a Department of Education GAANN fellowship and by the National Science Foundation (CCR-9811164 and CCR-0203813).
FV is also with the Center for Embedded Computer Systems at the University of California, Irvine.
Authors' address: Department of Computer Science and Engineering, University of California, Riverside, CA 92521; email: gstitt@cs.ucr.edu.
Permission to make digital/hard copy of part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date of appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permision and/or a fee.
© 2004 ACM 1539-9087/04/0200-0218 $5.00

and Wawrzynek 1997), as well as reduction in power or energy [Henkel and Li 1998; Henkel 1999; Stitt et al. 2002; Wan et al. 1998]. These advantages are gained at the expense of increased silicon area—area that is becoming cheaper and more readily available every year.

Many previous efforts have focused on partitioning an application that consists of numerous concurrent processes [Hou and Wolf 1996; Kalavade and Lee 1994]. Our work focuses on partitioning a single sequential program among a microprocessor and one or more custom coprocessors. In such single-program partitioning, custom coprocessors execute certain functions that were previously implemented in software. Such partitioning is possible in embedded systems, where the program is often fixed for the lifetime of the system. Previous work on single-program partitioning [Balboni et al. 1967; Eles et al. 1997; Gajski et al. 1998; Henkel and Ernst 1977; Vanmeerbeeck et al. 2001] has emphasized exploration of large numbers of candidate partitionings in order to meet timing constraints, utilizing powerful search algorithms such as simulated annealing, and utilizing sophisticated estimation models.

However, we have observed that most embedded applications spend a majority of their time in a few small loops or subroutines. Therefore, we will discuss a straightforward methodology for hardware/software partitioning that capitalizes on this observation, and is easy to implement manually or automatically.

The primary contribution of our work, though, is an extensive examination of the energy savings as well as speedups possible through hardware/software partitioning. We have examined numerous benchmarks ranging from small applications from the PowerStone suite [Malik et al. 2000], to medium-sized applications from MediaBench [Lee et al. 1997] and NetBench [Mernik et al. 2001]. We have utilized an 8-bit microcontroller as well as 32-bit processors. We have analyzed energy savings by using estimation models, as well as by taking physical measurements of real platforms. We have examined partitioning for an application-specific integrated circuit (ASIC) design flow, as well as for increasingly popular single-chip platforms having a microprocessor plus configurable-logic [Altera Corporation 2001; Atmel FPSLIC; E5; Triscend Corporation; Xilinx Corporation]. We have considered energy savings of partitioning using a microprocessor with low-power standby mode, and with a voltage-scalable power source.

2. HARDWARE/SOFTWARE PARTITIONING METHOD

2.1 Problem Description

We assume a designer is interested in reducing the energy required by a software application, in speeding up the application, or both. We assume the common situation in embedded systems of an application being made up of a particular repeating task. In some cases, that task must repeat once every X seconds; executing more frequently is not necessary. For example, an audio decompressor might have to decode a compressed audio frame once every 2 ms in order to provide a steady audio stream. In other cases, we may want the task to execute as frequently as possible. Thus, our goal is to reduce the energy for each

execution of the task, or to just speedup the task. Energy, measured in joules, is the product of time (seconds) and power (watts). In general, hardware/software partitioning reduces energy by reducing the execution time of the task, that is, by speeding up the task. Speedup is defined as the old execution time (software only) divided by the new execution time (software and hardware). The speedup typically occurs because a custom coprocessor can often execute a software region in one clock cycle that would have required numerous assembly instructions in software, due to the fine-grained parallelism possible in custom hardware. For example, the following software:

$$\begin{aligned}&\text{if } (a + b < c + d)\\&\quad x = y + z;\\&\text{else if } (a * b > e - f)\\&\quad y = x + z * 5;\end{aligned}$$

might require dozens of clock cycles in software, but could easily be accomplished in one cycle using custom hardware.

However, energy reduction is not obvious from this speedup because such partitioning typically increases the power of the system while the system is executing. Thus, to reduce energy, the speedup must be great enough to overcome the increase in power.

2.2 Critical Loop Detection

Past work in hardware/software partitioning, including our own, has typically emphasized extensive exploration of the partitioning solution space. Exploration algorithms typically examine thousands of possible partitionings. However, during our experiences with embedded applications, we have observed that most applications spend a majority of their time in just a few loops or subroutines—what we will call *critical loops*. We use the term critical loop for any loop that accounts for roughly 7% or more of a task's execution time. Though we use the term "critical loop" for simplicity, sometimes the region actually represents a subroutine. That subroutine is usually critical due to being called from a loop or due to containing a loop, so the term "loop" is appropriate. In very few cases, the subroutine is critical because of being called from numerous places throughout a program.

The number of critical loops in each benchmark is typically between two and four. Amdahl's law [Amdahl 1967] leads us to realize that we should focus initially on those critical loops to obtain our speedup. For example, a task may have a critical loop that accounts for 60% of the task's execution time, and numerous other loops that each account for only 5% each. Speeding up the critical loop may ideally result in a speedup of 100/(100 – 60) = 2.5, whereas speeding up any of the other loops could have at best only resulted in a speedup of 100/(100 – 5) = 1.1; even speeding up all of those other loops would have at best resulted in a speedup of 100/(100 – 40) = 1.7.

Table I summarizes critical loop statistics for a variety of benchmarks on several different microprocessors. The prefixes PS, MB, or NB indicate whether the benchmark application was taken from PowerStone [Malik et al. 2000],

Table I. Critical Loops for the Benchmarks Studied

			Critical Loops											
			Size				% time				Ideal Cum. Speedup			
Benchmark	Arch	Size	L1	L2	L3	L4	L1	L2	L3	L4	L1	L2	L3	L4
PS_g3fax	I8051	8,270	24				55%				2.2			
PS_crc	I8051	810	58				62%				2.6			
PS_summin	I8051	1,593	130	128			14%	6%			1.2	1.3		
PS_brev	I8051	2,406	1710				93%				14.3			
PS_matmul	I8051	836	212				85%				6.7			
PS_g3fax	MIPS	4,452	24				31%				1.4			
PS_adpcm	MIPS	7,640	88	64			17%	13%			1.2	1.4		
PS_crc	MIPS	4,288	68				65%				2.9			
PS_des	MIPS	6,116	360				52%				2.1			
PS_engine	MIPS	4,432	32	32			14%	14%			1.2	1.4		
PS_jpeg	MIPS	5,960	116				10%				1.1			
PS_summin	MIPS	4,136	48	52			36%	12%			1.6	1.9		
PS_v42	MIPS	6,388	60				23%				1.3			
PS_brev	MIPS	3,968	416				70%				3.3			
MB_g721	SS	11,878	31	594			45%	10%			1.8	2.2		
MB_adpcm	SS	9,302	153				99.9%				1000.0			
MB_pegwit	SS	24,990	62	62	64	31	35%	35%	4%	3%	1.5	3.3	3.8	4.3
NB_dh	SS	21,678	100	77	73		40%	18%	17%		1.7	2.4	4.0	
NB_md5	SS	10,724	5	18	850		13%	11%	32%		1.1	1.3	2.3	
NB_tl	SS	12,140	9				51%				2.0			
NB_url	SS	13,526	17				80%				5.0			
% of benchmark size:			2%	3%	3%									
% of benchmark time:							47%	62%	80%					

MediaBench [Lee et al. 1997; MediaBench], or NetBench [Mernik et al. 2001]. *Arch* indicates the microprocessor architecture onto which we compiled the application, being either an Intel 8051 8-bit microcontroller, a MIPS 32-bit embedded processor [MIPS], or the SimpleScalar (SS) processor (a MIPS extension) [Burger and Austin 1997]. *Size* indicates the static size in bytes of the application after being compiled to the given architecture.

The *Critical Loops* columns provide statistics on the most critical loops of each application, up to four of them (L1, L2, L3, and L4). *Size* represents the static size of the loop in bytes. *% time* indicates the percentage of task execution time (in this case, the percentage of total cycles) that this loop accounts for. *Ideal Cum. Speedup* represents the speedup that would ideally be obtained if this loop were executed in zero time. The speedups shown are cumulative for each successive loop, so the speedup under loop L2 assumes both L1 and L2 execute in zero time.

We obtained the data in Table I as follows. After compiling the C source code of the benchmark to a binary, for the 8051, MIPS, and SimpleScalar, we executed the binary on a cycle-accurate instruction set simulator. We developed our own instruction-set simulators for the 8051 [Univ. of California] and for the MIPS [Givargis et al. 2001], and we used the SimpleScalar simulator for the SimpleScalar processor. We configured each simulator to output an instruction trace. We also wrote a tool that parses each binary and outputs a listing of all the loops and subroutine locations. We then created a tool, called LOOAN

[Villarreal et al. 2001], that reads the loop/subroutine file, and then processes the instruction trace, maintaining a wide variety of statistics on loop/subroutine behavior, including the number of visits to the loop/subroutine and the number of iterations per loop—keeping minimums, maximums, and averages of these numbers. While this tool was quite useful in isolating critical loops, the trace files the tool generates are rather large, exceeding several gigabytes in some cases. Such size not only results in long run times for LOOAN, but can also exceed available disk space. Thus, we plan to update the instruction-set simulators to keep LOOAN's statistics during runtime (related work at UCR has already resulted in integration of LOOAN with the Simics simulator [Werner and Magnusson 1997]).

Table I indicates the extent to which most execution time is spent in just a few small loops. Many people refer to this phenomenon as the "80–20 rule" or the "90–10 rule," wherein software spends 90% of the time in 10% of the code. From the averages at the bottom of the table, we see that the phenomenon in these benchmarks results in what we might call a "50–2 rule"—about 50% of the time is spent in 2% of the code, or an "80–3 rule"—about 80% of the time is spent in 3% of the code.

The main observation we might make from these data is that most of our speedup will come from speeding up just one to three very small loops. This observation implies that an extensive solution space search during hardware/software partitioning is not necessary. This observation may also imply that only a small amount of hardware may be necessary to gain speedups of 2 to 3, and that in some cases, extremely high speedups may be possible with that small amount of hardware. With such speedups, we may also find good energy savings.

2.3 Partitioning Approach

To examine different hardware/software partitions of the applications, we began by ordering the critical loops according to their percentage of total time, with the loops labeled L1, L2, and so on. We then created a version of the application with all the critical loops moved to hardware. We modeled the hardware using a synthesizable register–transfer level VHDL process [Synopsys]. Each process described a finite-state machine, where we scheduled C-level statements into a minimal number of states.

When multiple loops from the application were synthesized, we included them as substates of a single-state machine, so that when synthesized they would share hardware. Such sharing was possible because we were guaranteed that the loops would not execute concurrently to one another, since they came from sequential software.

For our microprocessor/configurable-logic experiments, our target architecture was based on that in Figure 1, which is similar to the architecture found in Triscend's single-chip microprocessor/configurable-logic platforms [Triscend Corporation]. Unlike Triscend's platforms, we assume the configurable system logic (CSL) is a master of the bus. CSL bus mastering allows for more flexible CSL memory accesses compared to a DMA. However, a DMA can generally

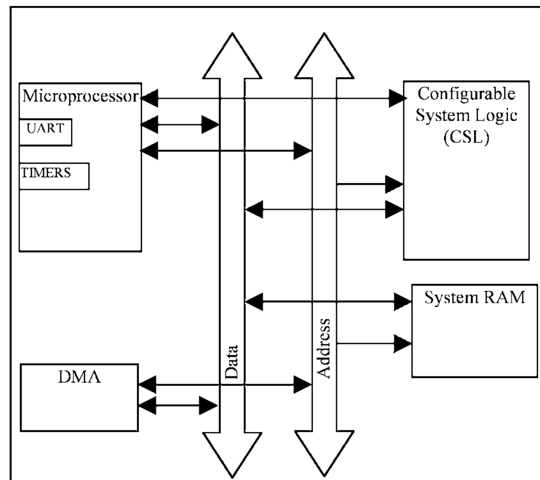

Fig. 1. Target architecture for single chip platform microprocessor with configurable logic.

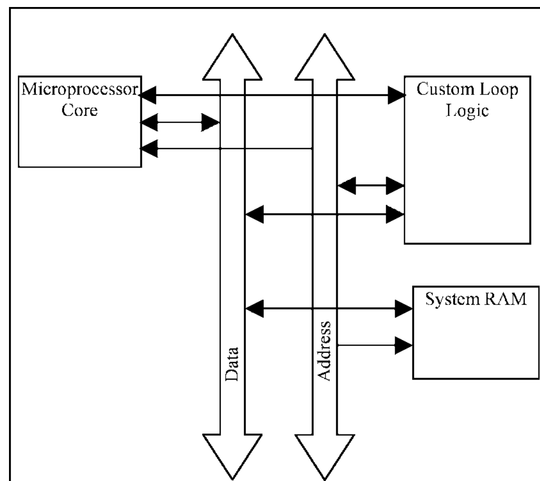

Fig. 2. Target architecture for single chip ASIC microprocessor core with custom loop logic.

handle block accesses more efficiently and allows for parallel execution of the processor and CSL. The added flexibility of giving the CSL the option of using a DMA or directly accessing the bus allows for hardware to be implemented for almost any software region. Communication between the microprocessor and CSL takes place via shared memory and several direct signals. Some of this shared memory is implemented in registers in the CSL, which have a direct connection to the hardware. The processor can efficiently communicate with the hardware by writing or reading from these registers. Due to a variety of types of on-chip configurable logic (FPGA, CPLD, and PLA), we use the more general term CSL to refer to all of these.

For the core-based ASIC experiments, we use a simpler architecture shown in Figure 2. This model shares a memory between the microprocessor and custom

hardware, but does not require a DMA component because the execution of the processor and custom logic is guaranteed not to overlap. The simplified architecture also has less interconnect between the microprocessor and custom logic because of the absence of the CSL.

We implemented each partitioning by replacing the selected software regions with a handshaking routine. The software would activate the custom hardware (either in the CSL or on the ASIC) using a start signal. For the microprocessor, such activation consists of simply setting a memory-mapped register with a direct connection to the hardware. The microprocessor then enters a low-power state while waiting for the hardware, achieved by setting a bit in a special function register. The processor then waits for the hardware to assert an interrupt, thereby waking up the processor. While the software partition is running, the hardware partition enters a low-power idle state. Waking up the processor from its low-power state requires anywhere between a few cycles to a few dozen, depending on the processor. In either case, these cycles are only expended after a loop in hardware completes (and not on every iteration), and thus is generally negligible compared to the thousands of cycles for the loop execution.

Currently, we are only considering the situation where the microprocessor and CSL execute in mutual exclusion. Mutually exclusive execution simplifies the architecture because the microprocessor and CSL will never access memory at the same time. In fact, because we are partitioning a sequential program, there would likely be little benefit to parallelizing the execution of the hardware and software partitions.

3. SPEEDUP AND ENERGY SAVINGS FOR MICROPROCESSOR/CSL DEVICES

3.1 Estimation Based

In Section 2.2, we discussed our method for determining software cycles, utilizing instruction-set simulators. To determine the software cycles for a partitioned design, we replaced the software loop with the required handshaking behavior. To determine the hardware cycles for a loop, we pessimistically assumed the longest path through the loop body was taken for every loop iteration. Such an assumption enabled us to avoid having to get every example simulating perfectly—something not necessary to estimate the speedup and energy improvements. Thus, the actual speedups and energy savings from partitioning may actually be slightly better than what we report.

Speedup data is shown in Table II for the benchmarks on which we performed estimation. $Cycles_{orig}$ represent the clock cycles for the entire application executing on a microprocessor. For the critical loops, $Cycles_{sw}$ represent the cycles those loops account for on the microprocessor, while $Cycles_{hw}$ represent the number of clock cycles needed for those loops to execute in the CSL hardware. We see that the speedups (Sp.) range from almost none (1.1) to 12.9, averaging 3.2. For all examples, the MIPS ran at 100 MHz and the 8051 ran at 25 MHz. The custom hardware in the CSL was run at the maximum possible clock frequency

Table II. Estimated Microprocessor/CSL Speedups

Example	Arch	Loop Performance				
		$Cycles_{orig}$	$Cycles_{sw}$	$Cycles_{hw}$	Clk_{hw}	Sp.
PS_g3fax	8051	19,675,456	10,812,544	176,562	25	2.2
PS_crc	8051	291,196	180,224	7,168	25	2.5
PS_summin	8051	109,821,892	20,394,080	384,416	25	1.2
PS_brev	8051	330,064	305,768	1,360	25	12.9
PS_matmul	8051	119,420	101,576	2,560	25	5.9
PS_g3fax	MIPS	15,600,000	4,720,000	599,000	100	1.4
PS_adpcm	MIPS	113,000	29,300	5,440	100	1.3
PS_crc	MIPS	5,040,000	3,480,000	460,800	100	2.5
PS_des	MIPS	142,000	70,700	15,100	100	1.6
PS_engine	MIPS	915,000	145,000	28,100	100	1.1
PS_jpeg	MIPS	7,900,000	646,000	171,000	100	1.1
PS_summin	MIPS	2,920,000	1,270,000	266,000	100	1.5
PS_v42	MIPS	3,850,000	846,000	216,000	100	1.2
PS_brev	MIPS	3,566	2,499	138	100	3.0
MB_g721	MIPS	838,230,002	457,674,179	9,985,261	100	2.1
MB_adpcm	MIPS	32,894,094	32,866,110	1,183,260	42	11.6
MB_pegwit	MIPS	42,752,919	33,276,287	2,167,651	50	3.1
NB_dh	MIPS	1,793,032,157	1,349,063,192	45,156,767	69	3.5
NB_md5	MIPS	5,374,034	3,046,881	289,877	47	1.8
NB_tl	MIPS	57,412,470	29,244,221	2,479,552	58	1.8
					Average:	3.2

reported by CSL tools after synthesis, place and route. These frequencies are specified in the column labeled Clk_{hw}.

We used Xilinx's virtex power estimator [Virtex] to estimate the power of the CSL executing the critical loops of each example, for a 0.18 μm FPGA technology at 1.8 V (in particular, for the XCV50E), shown in Table III as P_{hw}. We must also consider that the idle microprocessor continues to consume power while the CSL is active. Through physical measurement of Triscend's parts, we determined the idle microprocessor to consume 85% of the power of its active state, so we use 85% throughout the experiments.

We used typical power values for a commercial MIPS processor [MIPS Technologies] in 0.18 μm CMOS at 1.8 V, shown as P_{sw}, for the microprocessor active state. However, as above, we must also consider the idle power of the CSL while the processor is active, which we found through experimentation with Triscend parts to be about 12.5% of the CSL active power.

When either the microprocessor or CSL is active, we must also consider the power of interconnect and memory, P_i, which we obtained through physical measurement on Triscend parts and used throughout the experiments. Thus, we developed the following equation for total system energy E:

$$E = Time_{sw} * (P_{sw} + 0.125 * P_{hw} + P_i) + Time_{hw} * (0.85 * P_{sw} + P_{hw} + P_i).$$

$Time_{sw}$ is the number of cycles the microprocessor is active times the microprocessor clock period. We multiply $Time_{sw}$ by the system power consumed while the microprocessor is active. Similerly, $Time_{hw}$ is the number of cycles the CSL is active times the CSL clock period, which is then multiplied by the system power consumed while the CSL is active.

Table III. Estimated Microprocessor/CSL Power, Energy, and Area

Example	Arch	Power		Energy			Area
		P_{sw}	P_{hw}	E_{orig}	$E_{sw/hw}$	E_{Sav}	
PS_g3fax	8051	0.05	0.032	0.1142	0.05408	53%	2,858
PS_crc	8051	0.05	0.028	0.0017	0.00071	58%	770
PS_summin	8051	0.05	0.033	0.6376	0.53657	16%	4,191
PS_brev	8051	0.05	0.034	0.0019	0.00015	92%	3,961
PS_matmul	8051	0.05	0.035	0.0007	0.00012	82%	5,882
PS_g3fax	MIPS	0.07	0.111	0.0265	0.02163	18%	2,858
PS_adpcm	MIPS	0.07	0.181	0.0002	0.00018	6%	8,075
PS_crc	MIPS	0.07	0.061	0.0086	0.00379	56%	770
PS_des	MIPS	0.07	0.197	0.0002	0.00019	20%	9,031
PS_engine	MIPS	0.07	0.082	0.0016	0.00146	6%	2,074
PS_jpeg	MIPS	0.07	0.092	0.0134	0.01360	−1%	3,161
PS_summin	MIPS	0.07	0.111	0.0050	0.00375	24%	4,191
PS_v42	MIPS	0.07	0.102	0.0065	0.00605	7%	3,319
PS_brev	MIPS	0.07	0.107	0.0000	0.00000	62%	3,961
MB_g721	MIPS	0.07	0.152	1.4250	0.75035	47%	5,811
MB_adpcm	MIPS	0.07	0.130	0.0559	0.00821	85%	14,132
MB_pegwit	MIPS	0.07	0.170	0.0727	0.03241	55%	18,150
NB_dh	MIPS	0.07	0.121	3.0482	1.00547	67%	21,383
NB_md5	MIPS	0.07	0.251	0.0091	0.00722	21%	90,074
NB_tl	MIPS	0.07	0.059	0.0976	0.05930	39%	5,478
					Average:	34%	10,507

Energy results are shown in Table III. E_{orig} is the energy for the unpartitioned, software-only application, while $E_{sw/hw}$ is the energy after partitioning. In general, we see modest energy savings, averaging 34%, due to the lack of major speedups. These speedups were obtained using an equivalent of 10,507 logic gates on average.

3.2 Physical Measurement Based

We obtained two single-chip microprocessor/CSL devices from Triscend: an E5 and an A7. The E5 contained an accelerated 8051 8-bit microcontroller, which used only four clock cycles per instruction byte instead of the typical 12 cycles per instruction, and a 40,000 gate equivalent CSL. The clock frequency for both the 8051 and the CSL was 25 MHz. The A7 contained an ARM7 32-bit microprocessor plus a 40,000 gate equivalent CSL, both of which were clocked at 40 MHz. We used these parts to determine energy and speedup through physical measurement rather than estimation. We connected a digital multimeter to each device to measure current, and we used the timer available on our workstation to measure time. By multiplying current with voltage, we obtained power, which could be multiplied by our time measurements to obtain energy.

Results for the benchmarks we implemented on the A7 and E5 are shown in Table IV. As getting complete working implementations was rather time consuming, we obtained results for a subset of the benchmarks. We see good speedups and energy savings. We also see that our estimated results were reasonably accurate, and that our energy estimates were perhaps even a bit conservative.

Table IV. Microprocessor/CSL Speedups and Energy Savings from Physical Measurements of Triscend A7 and E5 Devices

Benchmark	$Time_{orig}$	$Time_{sw/hw}$	Sp.	P_{orig}	$P_{sw/hw}$	E_{orig}	$E_{sw/hw}$	E_{sav}
A7 Results								
PS_g3fax	11.47	7.44	1.5	1.320	1.332	15.140	9.910	35%
PS_crc	10.92	4.51	2.4	1.320	1.320	14.414	5.953	59%
PS_brev	9.84	3.28	3.0	1.332	1.344	13.107	4.408	66%
		Average	2.3				Average	53%
E5 Results								
PS_g3fax	15.16	7.11	2.1	0.252	0.270	3.820	1.920	50%
PS_crc	10.64	4.64	2.3	0.207	0.225	2.202	1.044	53%
PS_brev	17.81	1.81	9.8	0.252	0.270	4.488	0.489	89%
PS_matmul	32.66	2.06	15.9	0.270	0.288	8.818	0.593	93%
		Average	7.5				Average	71%

Table V. Estimated Speedups for a Core-Based ASIC

		Loop Performance				
Example	Arch	$Cycles_{orig}$	$Cycles_{sw}$	$Cycles_{hw}$	Clk	Sp.
PS_g3fax	8051	19,675,456	10,812,544	176,562	25	2.2
PS_crc	8051	291,196	180,224	7,168	25	2.5
PS_summin	8051	109,821,892	20,394,080	384,416	25	1.2
PS_brev	8051	330,064	305,768	1,360	25	12.9
PS_matmul	8051	119,420	101,576	2,560	25	5.9
PS_g3fax	MIPS	15,600,000	4,720,000	599,000	100	1.4
PS_adpcm	MIPS	113,000	29,300	5,440	100	1.3
PS_crc	MIPS	5,040,000	3,480,000	460,800	100	2.5
PS_des	MIPS	142,000	70,700	15,100	100	1.6
PS_engine	MIPS	915,000	145,000	28,100	100	1.1
PS_jpeg	MIPS	7,900,000	646,000	171,000	100	1.1
PS_summin	MIPS	2,920,000	1,270,000	266,000	100	1.5
PS_v42	MIPS	3,850,000	846,000	216,000	100	1.2
PS_brev	MIPS	3,566	2499	138	100	3.0
MB_g721	MIPS	838,230,002	457,674,179	9,985,261	100	2.1
MB_adpcm	MIPS	32,894,094	32,866,110	1,183,260	100	27.2
MB_pegwit	MIPS	42,752,919	33,276,287	2,167,651	100	3.7
NB_dh	MIPS	1,793,032,157	1,349,063,192	45,156,767	100	3.7
NB_md5	MIPS	5,374,034	3,046,881	289,877	100	2.1
NB_tl	MIPS	57,412,470	29,244,221	2,479,552	100	1.9
				Average		4.0

4. SPEEDUP AND ENERGY SAVINGS FOR CORE-BASED ASICs

The results given in the previous section utilize prefabricated single-chip microprocessor/CSL devices. We now describe estimated results if the critical loops could be implemented in custom hardware alongside a microprocessor core on a single ASIC. We utilized Synopsys synthesis and power estimation tools to obtain estimates for a 0.18 μm library similar to that used for the microprocessor.

The speedup results of hardware/software partitioning in an ASIC design are shown in Table V. Average speedup increased to 4.0, due to higher clock frequencies for the hardware partitions. Energy savings are shown in Table VI.

Table VI. Estimated Power, Energy, and Area for a Core-Based ASIC

Example	Arch	Power		Energy			Area
		P_{sw}	P_{hw}	E_{orig}	$E_{sw/hw}$	E_{sav}	
PS_g3fax	8051	0.05	0.001	0.11423	0.05249	54%	2,850
PS_crc	8051	0.05	0.001	0.00169	0.00068	60%	1,515
PS_summin	8051	0.05	0.000	0.63758	0.52150	18%	2,261
PS_brev	8051	0.05	0.001	0.00192	0.00015	92%	1,955
PS_matmul	8051	0.05	0.002	0.00069	0.00012	83%	3,989
PS_g3fax	MIPS	0.07	0.071	0.02652	0.02084	21%	2,850
PS_adpcm	MIPS	0.07	0.031	0.00019	0.00016	19%	10,592
PS_crc	MIPS	0.07	0.004	0.00857	0.00341	60%	1,515
PS_des	MIPS	0.07	0.021	0.00024	0.00015	38%	8,580
PS_engine	MIPS	0.07	0.010	0.00156	0.00137	12%	1,422
PS_jpeg	MIPS	0.07	0.006	0.01343	0.01267	6%	2,233
PS_summin	MIPS	0.07	0.003	0.00496	0.00324	35%	2,261
PS_v42	MIPS	0.07	0.009	0.00655	0.00550	16%	2,155
PS_brev	MIPS	0.07	0.002	0.00001	0.00000	66%	1,955
MB_g721	MIPS	0.07	0.002	1.42499	0.66403	53%	1,825
MB_adpcm	MIPS	0.07	0.029	0.05592	0.00228	96%	13,462
MB_pegwit	MIPS	0.07	0.047	0.07268	0.02113	71%	8,569
NB_dh	MIPS	0.07	0.051	3.04815	0.87815	71%	15,833
NB_md5	MIPS	0.07	0.028	0.00914	0.00458	50%	26,454
NB_tl	MIPS	0.07	0.011	0.09760	0.05249	46%	2,488
					Average	48%	5,738

The energy savings improve to nearly 50%. The reason for the increased energy savings is primarily because the hardware partition consumes nearly an order of magnitude less power in an ASIC than in CSL. The core-based ASIC required, on an average, only 5738 gates to implement the hardware partition.

5. VOLTAGE SCALING

Dynamic power consumption in CMOS designs is proportional to the supply voltage squared. Therefore, voltage scaling can be effective at reducing energy because lowering the voltage results in a quadratic reduction in power. Lowering voltage also increases the delay of the critical path, resulting in a slower clock and decreased performance. Due to the decreased performance, voltage scaling is typically performed dynamically during nonperformance critical parts of an application. An example of a voltage-scalable processor is the Intel XScale [Intel], having the capability to reduce the clock and voltage dynamically in order to reduce power at the expense of performance.

Hardware/software partitioning introduces a new possibility for voltage scaling. Due to the speedups achieved from the custom hardware, we can reduce voltage until the performance matches the original software, while consuming much less energy due to the approximately quadratic reduction in power.

We estimated the energy savings from voltage scaling using the following formulae [Gonzalez et al. 1997]:

$$T \alpha\, V/(V - V_t)^2$$
$$T = k * V/(V - V_t)^2,$$

where T is the delay of the critical path, V is the supply voltage, V_t is the threshold voltage, and k is a design-dependent constant.

Using the previous formulae, we are able to derive an equation for determining the clock frequency at a given supply voltage:

$$F = 1/T$$
$$F = (V - V_t)^2/(k * V),$$

where F is the clock frequency.

We first determine how much we can reduce the clock in order to match the performance of the software-only design. We then estimate the delay of the critical path by using the maximum clock frequency. Using this delay, we can determine k. We use a threshold voltage of 0.8 V. With this information, we can determine the minimum supply voltage that allows the design to run at the reduced clock speed. Once we have determined this voltage, we can estimate power for the voltage-scaled system in the following way:

$$C = P_o/\left(0.5 * V_o^2 * a * F_o\right)$$
$$P = 0.5 * V^2 * C * a * F$$
$$P = 0.5 * V^2 * \left(P_o/\left(0.5 * V_o^2 * a * F_o\right)\right) * a * F$$
$$P = \left(V^2/V_o^2\right) * (F/F_o) * P_o,$$

where P_o, V_o, and F_o are the power, voltage, and clock frequency of the system before voltage scaling, C is the capacitance of the system and a is the switching frequency. P, V, and F are the power, voltage, and clock frequency after voltage scaling. Therefore, we are estimating the power of the voltage-scaled system by using the power of the original system and the voltages and clock frequencies of both systems.

Results for voltage scaling are shown in Table VII. PSR is the percent speed reduction in order to match the original software. Clk_{VS} is the lower clock frequency used to achieve the required speed reduction. The 8051 had an original clock speed of 25 MHz and the MIPS had a clock of 100 MHz. V_{VS} is the voltage after voltage scaling. All examples originally used a supply voltage of 1.8 V. Power is the original power of the system. Power_{VS} is the power after voltage scaling. Energy is the amount of energy for each example on the original system. $\text{Energy}_{\text{VS}}$ is the energy required after voltage scaling. E_{sav} is the energy savings achieved from voltage scaling.

We see that voltage scaling increases the energy savings by nearly an additional 14%, to 62%.

6. CONCLUSIONS

Our experiments demonstrate that significant performance and energy benefits can be obtained for a wide variety of real software applications by moving just a small amount of critical code to hardware, while in some cases the speedups and energy savings can be huge. One interesting conclusion is that increasingly popular single-chip microprocessor/configurable-logic platforms can yield big improvements over microprocessor-only platforms, using

Table VII. Estimated Energy Savings for an ASIC Assuming a Voltage-Scalable Microprocessors

Example	Arch	PSR	Clk_{VS}	V_{VS}	Power	$Power_{VS}$	Energy	$Energy_{VS}$	E_{Sav}
PS_g3fax	8051	54%	11.5	1.39	0.145	0.040	0.11423	0.03119	73%
PS_crc	8051	59%	10.1	1.36	0.143	0.033	0.00169	0.00039	77%
PS_summin	8051	18%	20.4	1.68	0.145	0.103	0.63758	0.45392	29%
PS_brev	8051	92%	1.9	1.01	0.144	0.004	0.00192	0.00005	98%
PS_matmul	8051	83%	4.3	1.13	0.141	0.010	0.00069	0.00005	93%
PS_g3fax	MIPS	26%	73.6	1.61	0.168	0.099	0.02652	0.01547	42%
PS_adpcm	MIPS	21%	78.9	1.65	0.172	0.114	0.00019	0.00013	33%
PS_crc	MIPS	60%	40.1	1.34	0.159	0.035	0.00857	0.00178	79%
PS_des	MIPS	39%	60.8	1.51	0.167	0.071	0.00024	0.00010	58%
PS_engine	MIPS	13%	87.2	1.71	0.170	0.134	0.00156	0.00122	21%
PS_jpeg	MIPS	6%	94.0	1.75	0.170	0.151	0.01344	0.01191	11%
PS_summin	MIPS	34%	65.5	1.55	0.163	0.079	0.00496	0.00232	53%
PS_v42	MIPS	16%	83.6	1.68	0.168	0.122	0.00654	0.00471	28%
PS_brev	MIPS	66%	33.8	1.29	0.164	0.029	0.00001	0.00000	83%
MB_g721	MIPS	52%	48.4	1.41	0.178	0.053	1.50043	0.44280	70%
MB_adpcm	MIPS	91%	9.1	1.02	0.156	0.005	0.05888	0.00150	97%
MB_pegwit	MIPS	65%	34.8	1.30	0.183	0.033	0.07653	0.01419	81%
NB_dh	MIPS	69%	31.1	1.26	0.185	0.028	3.20953	0.50386	84%
NB_md5	MIPS	43%	56.8	1.48	0.179	0.069	0.00962	0.00370	62%
NB_tl	MIPS	40%	59.9	1.50	0.177	0.074	0.10277	0.04221	59%
								Average	62%

only a modest amount of hardware. A second conclusion is that good speedups can be obtained by moving just one to three small loops from software to hardware, for which extensive hardware/software exploration methods are not necessary. Thus, automation tools could foreseeably be developed whose main tasks would be profiling followed by synthesis of critical loops into hardware—something becoming increasingly possible with the advent of C-based synthesis tools.

We plan in the future to analyze the impacts of various architecture features, such as microprocessor and hardware clock frequencies and power consumption, interconnect power, CSL power, available hardware size, and memory bandwidth, on obtaining speedups and energy savings. We also plan to investigate the benefits of moving additional loops to hardware after the initial critical loops have been moved.

ACKNOWLEDGMENTS

We thank Brian Grattan for his assistance with partitioning and measurement of several of the benchmarks. We thank the Triscend Corporation for their donation of several boards to support our research, and for numerous technical discussions. This work was supported in part by the National Science Foundation (CCR-0203829) and a Department of Education GAANN fellowship.

REFERENCES

ALTERA CORPORATION. 2001. ARM-Based Embedded Processor PLDs.

AMDAHL, G. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings AFIPS 1967 Spring Joint Computer Conference 30*, 483–485.

ATMEL FPSLIC, http://www.atmel.com/atmel/products/prod39.htm.

BALBONI, A., FORNACIARI, W., AND SCIUTO, W. 1996. Partitioning and exploration in the TOSCA co-design flow. In *Proceedings of the International Workshop on Hardware/Software Codesign*, 62–69.

BURGER, D. AND AUSTIN, T. M. 1997. The SimpleScalar tool set, Version 2.0. In Tech. Rep. #1342, University of Wisconsin-Madison Computer Sciences Department.

E5 PRESS RELEASE, http://www.triscend.com/about/indexrelease051401.html.

ELES, P., PENG, Z., KUCHCINSKY, K., AND DOBOLI, A. 1997. System level hardware/software partitioning based on simulated annealing and tabu search. *Design Automation for Embedded Systems 2*, 1, 5–32.

GAJSKI, D.D., VAHID, F., NARAYAN, S., AND GONG, J. 1998. SpecSyn: An environment supporting the specify-explore-refine paradigm for hardware/software system design. *IEEE Transactions on VLSI Systems 6*, 1, 84–100.

GIVARGIS, T., VAHID F., AND HENKEL, J. 2001. System-level exploration for pareto-optimal configurations in parameterized systems-on-a-chip. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*.

GOKHALE, M. AND STONE, J. 1998. NAPA C: Compiling for hybrid RISC/FPGA architectures. *In Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*.

GONZALEZ, R., GORDON, B., AND HOROWITZ, M. 1997. Supply and threshold voltage scaling for low power CMOS. *IEEE Journal of Solid-State Circuits 32*, 8.

HAUSER, J. AND WAWRZYNEK, J. 1997. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, CA, 12–21.

HENKEL, J. 1999. A low power hardware/software partitioning approach for core-based embedded systems. In *Proceedings of the 36th ACM/IEEE Design Automation Conference*, 122–127.

HENKEL, J. AND ERNST R. 1997. A hardware/software partitioner using a dynamically determined granularity. In *Proceedings of the Design Automation Conference*.

HENKEL, J. AND LI, Y. 1998. Energy-conscious HW/SW-partitioning of embedded systems: A Case Study on an MPEG-2 Encoder. In *Proceedings of 6th International Workshop on Hardware/Software Codesign*, 23–27.

HOU, J. AND WOLF, W. 1996. Process partitioning for distributed embedded systems. In *Proceeding International Workshop on Hardware/Software Codesign*.

INTEL XSCALE PROCESSOR, http://developer.intel.com/design/intelxscale.

KALAVADE, A. AND LEE, E. 1994. A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem. In *Proceedings of the International Workshop on Hardware/Software Codesign*, 42–48.

LEE, C., POTKONJAK, M., AND MAGIONE-SMITH, W. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communication systems. In *Proceedings of MICRO*.

MALIK, A., MOYER, B., AND CERMAK, D. 2000. A low power unified cache architecture providing power and performance flexibility. In *Proceedings of the International Symposium on Low Power Electronics and Design*.

MEDIABENCH. http://www.cs.ucla.edu/~leec/mediabench/.

MERNIK, G., MANGIONE-SMITH, W. H., AND HU, W. 2001. NetBench: A benchmarking suite for network processors. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, 39–42.

MIPS TECHNOLOGIES, INC., http://www.mips.com.

STITT, G., GRATTAN, B., VILLARREAL, J., AND VAHID, F. 2002. Using on-chip configurable logic to reduce embedded system software energy. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA.

SYNOPSYS, http://www.synopsys.com.

TRISCEND CORPORATION, http://www.triscend.com. 2002.

UNIVERSITY OF CALIFORNIA, Riverside; Dalton Project. http://www.cs.ucr.edu/~dalton.

VANMEERBEECK, G., SCHAUMONT, P., VERNALDE, S., ENGELS, M., AND BOLSENS, I. 2001. Hardware/software partitioning of embedded system in OCAPI-xl. In *Proceedings of the International Symposium on Hardware/Software Codesign*, 30–35.

VILLARREAL, J., LYSECKY, R., COTTERELL, S., AND VAHID, F. 2001. Loop analysis of embedded applications. In Tech. Rep. UCR-CSE-01-03, University of California, Riverside.

Virtex Power Estimator, http://support.xilinx.com/cgi-bin/powerweb.pl.
Wan, M., Ichikawa, Y., Lidsky, D., Rabaey, J. 1998. An energy conscious methodology for early design exploration of heterogeneous DSPs. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, 111–117.
Werner, B. and Magnusson, P. 1997. A hybrid simulation approach enabling performance characterization of large software systems. In *Proceedings of MASCOTS*.
Xilinx Corporation. 2002. *Virtex-II Pro Platform FGPA Handbook*.

Received February 2002; revised August 2002; accepted April 2003