Journal of VLSI Signal Processing 2006

DOI: 10.1007/s11265-006-8538-6

Area and Power Reduction of Embedded DSP Systems using Instruction Compression and Re-configurable Encoding*

SUBASH CHANDAR AND MAHESH MEHENDALE
Texas Instruments (India) Pvt. Ltd., No. 66/3 Byrasandra, Bagamane Tech Park, Bangalore, 560 093, India

R. GOVINDARAJAN
Supercomputer Education & Research Centre, Department of Computer Science & Automation, Indian Institute of Science, Bangalore, 560 012, India

Received: 10 January 2005; Revised: 22 December 2005; Accepted: 17 February 2006

Published online: 25 July 2006

Abstract. In embedded control applications, system cost and power/energy consumption are key considerations. In such applications, program memory forms a significant part of the chip area. Hence reducing code size reduces the system cost significantly. A significant part of the total power is consumed in fetching instructions from the program memory. Hence reducing instruction fetch power has been a key target for reducing power consumption. To reduce the cost and power consumption, embedded systems in these applications use application specific processors that are fine tuned to provide better solutions in terms of code density, and power consumption. Further fine tuning to suit each particular application in the targeted class can be achieved through reconfigurable architectures. In this paper, we propose a reconfiguration mechanism, called *Instruction Re-map Table*, to re-map the instructions to shorter length code words. Using this mechanism, frequently used set of instructions can be compressed. This reduces code size and hence the cost. Secondly, we use the same mechanism to target power reduction by encoding frequently used instruction sequences to Gray codes. Such encodings, along with instruction compression, reduce the instruction fetch power. We enhance Texas Instruments DSP core TMS320C27x to incorporate this mechanism and evaluate the improvements on code size and instruction fetch energy using real life embedded control application programs as benchmarks. Our scheme reduces the code size by over 10% and the energy consumed by over 40%.

Keywords: code compression, embedded DSP systems, energy reduction, re-configurable architecture

1. Introduction

In recent years, there has been a significant growth in consumer electronics and semiconductor industry. This phenomenal growth is driven by the increased market for electronic products such as cellular phones, pagers, personal digital assistants (PDA), portable digital audio players, and digital cameras. These are high volume products that are extremely cost sensitive and have stringent constraints on power consumption, while having certain performance requirement.

In embedded applications, program memory forms a significant part of the entire chip area. Hence,

*A preliminary version of this paper has appeared in the International Conference on Computer Aided Design (ICCAD-2001), San Jose, CA, November 2001.

reducing program memory size reduces the overall chip area which in turn reduces the cost. However, to reduce the software development time and also due to the increased complexity of the embedded software, embedded system programmers use high-level languages. This, in turn, results in the use of compiler-generated code in embedded applications that is much larger than hand-optimized code, putting additional pressure on the program memory. Several techniques to reduce code size have gained considerable attention in embedded applications. A few examples of such techniques are variable length instructions [32, 36], dual instructions sets [1, 13], and code compression [7, 14, 18, 21, 40].

This paper focuses on reducing code size, and hence the cost, as well as the power consumed in an the embedded system. We concentrate on embedded systems which use processor cores, typically Digital Signal Processors (DSPs). DSP architecture is optimized for the digital signal processing applications. More specifically, the instruction set encoding is optimized for code size using variable length instructions, with frequently occurring instructions in the application class having smaller opcode size. However, the frequently occurring instruction set is decided based on a set of applications, and hence there is certain compromise in meeting the varying requirements of different applications in the target class. Thus the encoding of individual instructions using variable length opcodes is to *collectively reduce* the code size across all the applications in the target class. Such a scheme suffers from the disadvantage that the final instruction encoding need not be optimal for *each* of those applications.

To handle varying requirements of different applications and to achieve very high code density requires super specialization or customization for each application. This would mean the need for specialized architectures for each application. But, it is not practical to have one architecture per application due to the high cost associated in productizing a brand new architecture. Reconfigurable systems achieve the benefits of super specialization through customizable hardware and without incurring the excessive design cost. Using reconfigurable schemes it is possible to re-assign smaller width opcodes to instructions which are frequently used in the given application to reduce the code size, and hence the size of program memory. This implies that there is a scope for reducing the code size further, if we allow some amount of customization in the choice of encodings for the instructions.

Static reconfiguration refers to the ability to reconfigure the hardware resources once at the beginning of execution of an application. During the execution of a program the configuration remains the same. *Dynamic reconfiguration* refers to the ability to do run-time reconfiguration. The configuration changes dynamically during the execution of the program to exploit varying hardware resource requirements of the program. There is a finite reconfiguration overhead associated with dynamic reconfiguration which should be taken into account while measuring the benefits of dynamic reconfiguration.

In this paper, we propose a common hardware mechanism, referred to as the *instruction re-map scheme*, that can be used for both code size reduction and power reduction. The scheme is based on an instruction remap table using which instruction encoding can be changed to reduce code size and power. The proposed re-map scheme is simple and can easily be fitted in the pre-decode stage without incurring any additional runtime overhead. Also, the proposed scheme can be used for both static or dynamic reconfiguration. Further, by careful encoding of remappable instructions with Gray codes, our scheme achieves power reduction by reducing the number of (bit) toggles in the instruction fetch. Reducing the power consumption of instruction fetch is important as the instruction fetch stage is known to contribute more than 15% of the total power consumed [24].

We evaluate the proposed instruction remap scheme using real life application programs which are already optimized for code size. The instruction remap scheme achieves a code size reduction of over 10% on already optimized real life embedded control applications. The scheme also results in an energy reduction of more than 40% of the instruction fetch power.

The rest of this paper is organized as follows. In Section 2, we present a discussion on related work. We describe the proposed reconfiguration scheme in the subsequent section. In Section 4, we present the details of a re-map table organization and explain how static and dynamic reconfiguration can be achieved using the re-map table. The improvement in code size due to reconfiguration are presented in Section 5. In Section 6, we discuss how the

instruction re-map scheme can achieve reduction in instruction fetch power. This section also reports the energy reduction in three real-life embedded control applications. Finally we provide concluding remarks in Section 7.

2. Related Work

In this section we discuss related work in code compression and power reduction.

2.1. Code Compression

Wolfe et al., proposed a novel RISC architecture that can execute programs stored in the memory in a compressed form [14, 40]. This scheme is applicable for systems with a cache. At compile time, the cache line bytes are Huffman encoded [9] compressed code is stored in program memory and decompression takes place when instruction blocks are brought into the cache subsystem. Other algorithms for Compressed Code RISC Processor are presented in [21]. In this work, two approaches—ISA independent method using statistical encoding and ISA specific method using dictionary—are presented. IBM's CodePack [11, 12] scheme, employed in embedded PowerPC systems, resembles compressed code RISC processor in that the decompression is part of memory subsystem and the CPU always sees uncompressed instructions. In contrast, in our instruction remap scheme, the fetch stage of the processor sees only the compressed instruction. This helps in reducing instruction fetch power significantly.

Dictionary based methods [7, 19] have been presented in the literature for compressing the code. In [7], Devadas et al., propose two methods for code size minimization. In their first method, frequently occurring sequences are extracted to form dictionary entries. Then these code sequences are replaced by calls to the corresponding dictionary entries. Their second method is more flexible and allows compressing any substring of the dictionary to be replaced by a *CALD* instruction—a new instruction added to the base processor that specifies the address as well as the number of instructions to be executed from the dictionary. This scheme requires certain hardware modifications and the addition of a new instruction (*CALD* instruction) to the instruction set architecture. A downside of dictionary based compression schemes is the introduction of program discontinuities, and consequently the control transfer overheads, due to call instructions. Further since the call instruction itself is of size 1 or 2 instruction words, to be beneficial, the compressed instruction sequences must be at least 2 or 3 instructions. Hence, this method misses an important compression opportunity of compressing patterns consisting of single instructions. In comparison our instruction compression scheme can efficiently compress single instruction sequences, and compression does not introduce any program discontinuity.

Lefurgy et al., present a dictionary based approach to reduce code size which they call a *Compressed Program Processor* [19]. In this method, commonly occurring patterns are replaced by code words of shorter length, in the program memory. The code word fetched from the program memory is used to index into the dictionary to get the uncompressed instruction. A disadvantage of this method is that compressed instructions are also stored in the program memory, it introduces additional delay in the CPU-Memory interface path.

A compression method [17] used in the SHARC DSP processor identifies unique instruction words in the program and stores them in an instruction table. Each instruction in the program is then replaced with an index into this table. Since the index is shorter than the instruction and also that the instruction table overhead is usually small compared to the program size, compression is achieved. This scheme, however, requires modification, in the form of an additional pipeline stage, to the SHARC pipeline. Further this method requires an additional external memory bus to support simultaneous access to instruction table, program memory, and data memory. This scheme, like our remap scheme, achieves single instruction compression.

More recently, Menon and Shankar propose an ISA based code compression scheme [26]. The basic idea of their scheme is to divide the instructions into different logical classes and to build multiple dictionaries for them. They have evaluated their approach for ARM and TI's TMS320C62x processors. However, they do not consider power reduction. In [3], a family of dictionary-based code compression schemes have been proposed based on the concepts of static and dynamic entropy. The static and dynamic entropy discussed in the paper is similar to instruction frequency based on static and dynamic instruction counts in our scheme. One of their schemes uses a (relatively) large dictionary, a

part of which is stored in the memory. Although this work does address power reduction, it does not consider dynamic reconfiguration.

Classical compiler optimizations such as strength reduction, dead code elimination, and common sub-expression elimination have been used in [5, 6] to reduce code size. More recently, Lau et al., proposed a code compression scheme using *echo* instructions and their variants [15]. With echo instructions, two or more similar, but not necessarily identical, sections of code can be reduced to a single copy of the repeating code. Using mask registers and other compiler techniques such as register renaming and instruction scheduling, they enhance the scope of echo instructions for achieving better compression ratio. In [29], the authors present a code compression strategy based on control flow graph (CFG) representation of the embedded program. The basic idea is to keep all basic blocks compressed, and decompress only the blocks that are predicted to be needed in the near future. Their approach also decides the point at which the block could be compressed. The compression scheme is specifically meant to address memory space constraints rather than power reduction.

An LZW-based compression method for VLIW architectures was proposed in [23] to compress variable sized instruction blocks called branch blocks. Their approach generates the coding table on-the-fly during compression and decompression. However, the scheme compresses blocks of instructions as opposed to our approach which compresses individual instructions to small instruction width. Piguet et al., propose a code compression scheme with instruction by instruction decompression at run time, with the possibility to modify the program with the same compressed physical memory [30]. A class-based dictionary scheme was proposed in [27] where the processor remains unaware of the compression. Apart from reducing the code size, the proposed scheme results in reduction in power consumption and improvement of performance brought about by instruction cache compression. However no experimental evaluation of the scheme was discussed in [27].

ARM and MIPS processors use dual instruction sets—a 16-bit instruction set and a 32-bit instruction set—to achieve both code size and performance efficiency. Thumb has a subset of ARM instruction set defined as 16-bit instructions set [37]. Similarly MIPS-16 has a 16-bit instruction set that is a subset of MIPS-III instruction set [13]. While the 16-bit instruction set gives code size advantage, the 32-bit instruction set gives performance advantage. Time critical part of the code can use 32-bit instruction set mode for better performance while the rest of the code can use 16-bit instruction set mode to reduce code size. Although the 16-bit instruction sets achieve 30–40% code size reduction, they incur a performance penalty (15–20%) due to lower expressive power of 16-bit instruction sets.

Unlike any of the above approaches, we use dynamic (multiple) instruction remapping to exploit varying instruction usage within an application to get greater benefits.

2.2. Power Reduction

Gray coded addressing has been presented in the literature [28, 34] as a technique for reducing power in the program address bus of a processor. Gray coding is particularly attractive since it guarantees single bit transitions when consecutive addresses are accessed. This technique requires additional circuitry for encoding (binary-to-gray) and decoding (gray-to-binary). There is a power overhead due to this additional circuitry.

The power dissipation in the busses can be reduced even by using the *asymptotic zero-transition encoding*, referred to as *T0 coding* in [2]. This scheme is based on the observation that the values transmitted on the program address bus are often consecutive. This scheme exploits this property to avoid the transfer of consecutive addresses on the bus. For this purpose, a redundant line *INC* is added to the address bus. This line is used to transfer to the receiving subsystem the information about the sequentiality of the addresses. When two addresses to be transmitted are consecutive, the *INC* line is set to 1 and the address lines are frozen to avoid unnecessary transitions. The new address is computed by the receiver by incrementing the previous address value. When two addresses are not consecutive, then the *INC* line is set to 0 and the new address is explicitly transmitted on the address bus. With this scheme it is possible to achieve zero transitions (toggles) during the period when consecutive addresses are transmitted.

Bus invert coding [33] can be effectively used to reduce power dissipation in data busses. In this scheme, if the Hamming distance between successive data values is larger than $N/2$ where N is the bus width, then the current data value is inverted and

transmitted. If the Hamming distance is less than $N/2$, then the data value is transmitted in the original form. Thus this schemes reduces the maximum number of transitions on the bus to N/2. An additional bit I is added to the bus to make this possible. *Partial Bus Invert Coding* technique [33] is based on the observation that Bus Invert Coding performs better for smaller width busses. In this scheme, wider busses are partitioned into several narrower sub-busses, and each of these sub-busses is coded independently with its own invert signal I.

Cold scheduling [34] is an instruction scheduling technique that schedules operations so as to minimize the Hamming distance between successive instructions. Hence the switching activity on program data bus is reduced thereby resulting in reduced power dissipation in fetching the instructions. Power dissipation is reduced by identifying instructions that are fetched consecutively in an application, and assigning opcodes to these instructions in such a way to reduce the toggle [25, 38].

Other techniques such as clock gating, power conservation modes, and application specific instruction sets for low power have been used to reduce power consumption. A detailed survey of different techniques targeting low power is presented in [4, 8, 31].

While most of the techniques discussed above target only power reduction, our instruction remapping scheme achieves both code size and power reduction simultaneously. The schemes proposed in [20] and [22] also achieve power reduction by code compression. Power reduction is achieved by reducing the number of memory accesses using code compression in [20]. The reduction in number of memory accesses results in reduction in total number of toggles on the program address bus in fetching the entire code. The impact of code compression and arithmetic coding (to reduce the number of toggles) on power dissipation are presented in [22]. Here coding is used to reduce the number of toggles on program data bus. Further, unlike in the previous work, our scheme supports dynamic remapping of instructions which allows the reuse of the same set of short code words to be efficiently mapped to different sets of instructions at different points of time.

Hiraki et al., discuss a low power processor architecture using a decoded instruction buffer [10]. In this approach, during the program execution, the decoded instructions are temporarily stored in a small RAM called decoded instruction buffer (DIB) inside the processor. For loops, from second iteration onwards the instruction fetch and decode stages are stopped and the decoded instructions are directly taken from DIB. Since in signal processing applications a major part of the execution time is spent in executing tight loops, a significant power saving can be achieved with this approach. The results indicate up to 40% reduction in power consumption. This approach however saves power only for loops. An important property of repeated sequences of instructions in the program, not necessarily contained within a loop, is missed out in this method. Our approach looks at the sequence of instructions across the program and tries to optimize the power spent in fetching the instructions. Also memory required for storing decoded values of instructions is much higher, increasing the area. This is not desirable for embedded applications where cost is a key careabout.

Lee et al., discuss an instruction fetch energy reduction technique using loop caches [16]. This approach is similar to the approach discussed in [10] in the sense that both the approaches try to reduce energy spent in executing tight loops. In this approach, a small cache called loop cache is implemented. This loop cache stores a few recently fetched instructions. While executing small loops (determined by a new instruction "sbb'') the main cache is shut off and the instructions are taken from the loop cache. This approach reduces the number of fetches to the main cache, thereby saving fetch energy. The results presented in this work indicate that the number of instruction accesses to the main cache is reduced by about 40%. The actual energy reduction achieved is not presented in this paper. Unlike this approach, our approach saves power by reducing the toggles (and the number of instruction fetches) for frequently fetched instruction sequences. Hence the energy saving in our approach is not just limited to loops.

3. Instruction Re-map Scheme

In a processor with variable length instructions, the encoding of instructions into codes of different lengths is done with the objective of collectively reducing the code size across all the applications in the target class. Each instruction is assigned with a fixed encoding. For reconfigurable instructions, the encodings can be altered. In other words, reconfigurable instructions can be re-mapped to different

encoded values. We refer this process of mapping an instruction to an encoded value as *configuring the instruction*. This configuration can be done either statically, once prior to running a given application (static reconfiguration), or dynamically on the fly, possibly many times while running a given application program (dynamic reconfiguration). An instruction remapped to a shorter length encoding is referred to as a *compressed instruction*. The proposed scheme can be applied to accomplish static or dynamic reconfiguration as described in the following subsections.

3.1. Static Reconfiguration

In a static reconfiguration, the configuration (instruction encoding) happens once, prior to running a given application. This gives the flexibility of fine-tuning the encodings of instructions to suit the instruction usage in a given application. In this method, we profile the (assembly) program of the targeted application and obtain instruction usage information. Based on this information the instruction encoding is done such that higher the frequency of occurrence of an instruction, smaller is the size of the encoded value assigned to it. With statically reconfigured instructions, the same processor can have its instructions mapped to different encodings across different applications. But within a given application, the encoding for each instruction is fixed.

3.2. Dynamic Reconfiguration

Inside each given application, the instruction usage may vary across different segments of the code. In dynamic reconfiguration, the instructions can be remapped to different sets of encodings on the fly, possibly many times, while running a given application. In this scheme, we divide the application code into several segments. We profile each segment individually to obtain instruction usage information within that segment. The instructions are then assigned appropriate encodings based on the instruction usage in each segment. Instruction encoding is done for each segment individually. Thus, within a given application the same instruction that occurs in different parts of the code may have different encodings. This scheme addresses the varying requirements inside a given application and hence can result in a smaller code size compared to static reconfiguration.

It should be noted here that in both static and dynamic reconfiguration schemes, the reconfiguration—what instructions are remapped and at what program points—is decided statically, and the process of remapping instructions is done dynamically, once at the beginning of the execution for static reconfiguration and multiple times for dynamic reconfiguration. In the following section we discuss how remapping of instructions to shorter length opcodes is achieved.

4. Instruction Re-map Table

To implement reconfiguration of instructions, we propose a mechanism—*Instruction Re-map Table (IRT)*. The *IRT* can be viewed as a register file. Each entry in this table can hold a valid uncompressed instruction of size equal to 16 bits; the entry has a unique address which forms the compressed representation. A pair of special instructions are added to the existing instruction set to allow reconfiguration of the IRT. Once an instruction is written into the IRT, it can be referenced by the address of its location. Since the width of the address, which is the compressed instruction, is smaller than that of the actual instruction, we achieve compression. The details are discussed in the following subsections.

4.1. Organization of Re-map Table

The instructions to be compressed are first written into locations $Loc1$ to $Locm$ in the instruction re-map table as shown in Fig. 1. Configuring the re-map table will be discussed in Section 4.3. After configuring the IRT, each of the locations in it holds a 16-bit instruction. The width of the address of these locations is equal to $log_2 m$ where m is the number of locations in the instruction re-map table. $addr1$ in the figure is the address of first location ($Loc1$), $addr2$ is the address of second location ($Loc2$), and so on. Each instruction that is written into the table can now be referenced by the address of its corresponding location. For example, an instruction that is written into location $Loc1$ can now be referenced by $addr1$.

For an IRT of size 128 entries, each address will be 7-bit wide. We add one extra bit to this address to indicate that it is a compressed instruction. This address $addr1$ along with the newly added bit, forms

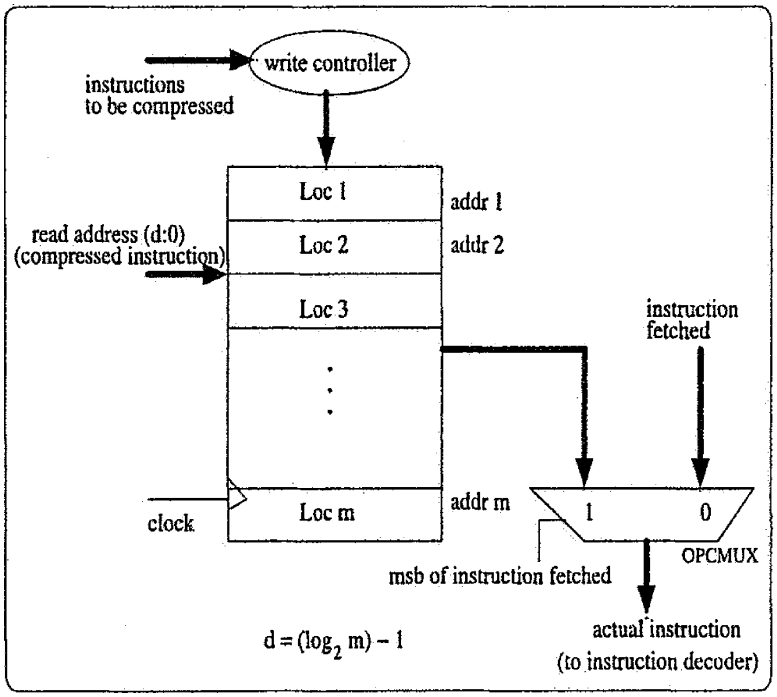

Figure 1. Instruction re-map table.

the compressed representation for the instruction that resides in $Loc1$ of the instruction re-map table. This added bit forms the most significant bit (msb) of the compressed instruction and is set to 1 to indicate that it is a compressed instruction. The *msb* of all uncompressed instructions are set to 0. That is, for all instructions, in the original (uncompressed) encoding the msb must be 0. If such a constraint cannot be met on the original encodings of the instructions, we can add an extra bit to the uncompressed instructions which will be set to 0. Due to this, the size of the uncompressed instructions increases by one bit. In our evaluations, we have conservatively included this overhead of one extra bit for uncompressed instructions. With this, the program memory is organized as 17-bit wide.

Now, the instructions that are written into the instruction re-map table have a unique compressed representation. All these instructions in the program memory are now replaced by their compressed representation. Other instructions appear as uncompressed. While encoding the instructions, the compressed and uncompressed instructions get their *msb* as 1 and 0, respectively.

Decompression of the compressed instruction is achieved as follows. The re-map table is indexed with $log_2 m$ bits following the most significant bit of the fetched instruction (compressed or uncompressed) as shown in Fig. 1. The most significant bit of the fetched instruction is used to select either the contents of the indexed location of the re-map table or the fetched instruction (without the most significant bit). Thus the output of the multiplexer is the uncompressed encoding of the fetched instruction.

Note that with the above compression scheme, instruction widths could become non-standard size. That is, uncompressed instructions are 17-bit wide while each compressed instruction could be 8-bit wide, for a re-map table size of 128 locations. Hence, this would require the instruction fetch logic to fetch several bytes at a time, and do appropriate bit re-aligning to get the instructions correctly. Since we are evaluating the potential with different re-map tables, we do not initially consider the problems with non-standard instruction widths and instruction re-alignment. However, we do report results, in Section 5.1.2, where each instruction is compressed as a 8-bit instruction (including the MSB which indicates

compressed instruction) for any re-map table of size less than or equal to 128 entries. With 8-bit compressed instructions, bit alignment problem simplifies as byte-alignment which can be easily handled.

In the above discussion, we have assumed the width of the re-map table as 16 bits. That is, each location stores a 16-bit encoding (uncompressed) of the instruction. Consequently only a 16-bit instruction can be compressed. If a 32-bit instruction appears frequently in the application, a simple solution to compress it is to consider it as two 16-bit instructions and compress each of them separately. Although this reduces the extent of compression, it retains the simplicity of the re-map table hardware.

4.2. Architectural Support for Instruction Re-map Table

We use Texas Instruments DSP core TMS320c27x [36] as a representative architecture to illustrate how the re-map table can be integrated with the architecture. TMS320c27x DSP core has eight pipeline stages, viz., Initiate-Fetch, Complete-Fetch, Predecode, Decode, Initiate-Read, Complete-Read, Execute and Write stages as shown in Fig. 2.

The address of the instruction to be fetched from program memory is made available on the address bus in Initiate-Fetch stage of the pipeline. During Complete-Fetch stage, the memory drives the instruction on the program data bus. The fetched instruction is decoded in two stages, namely predecode and decode. If the instruction needs data from data memory, the same is fetched with a two stage data memory read. Then it is executed in the Execute stage and the results are stored in writeback stage of the pipeline.

We enhance the DSP core with the inclusion of the instruction re-map table. The instruction re-map table fits in the pre-decode stage of the pipeline as indicated in Fig. 2. Although the instruction re-map table can be implemented outside the core and fit in the CPU memory interface, in deep submicron designs, where the interconnect delay dominates memory access, access to re-map table would become the bottleneck. To avoid this, we include the re-map table in the predecode stage where the access time of small table sizes can be hidden. Our analysis in Section 5.4 reveals that the access time for a table of size up to 128 entries is small (less than 0.5 ns) and can be easily accommodated in the pre-decode stage of an embedded processor (which are typically clocked at 200–600 MHz) without incurring any additional speed penalty (refer to Section 5.4 for more details).

4.3. Configuring the Re-map Table

In this section we discuss how to configure the IRT. First, a set of instructions are identified for compression. The method used for identifying the instructions to be compressed depend on whether static or dynamic reconfiguration is used, and is described in Sections 5.1 and 5.2, respectively.

The set of instructions that are identified for compression are then written into the IRT. Writing into the IRT is facilitated by adding a pair of special instructions to the existing instruction set. We call these instructions *BCONF* and *ECONF* : *BCONF*

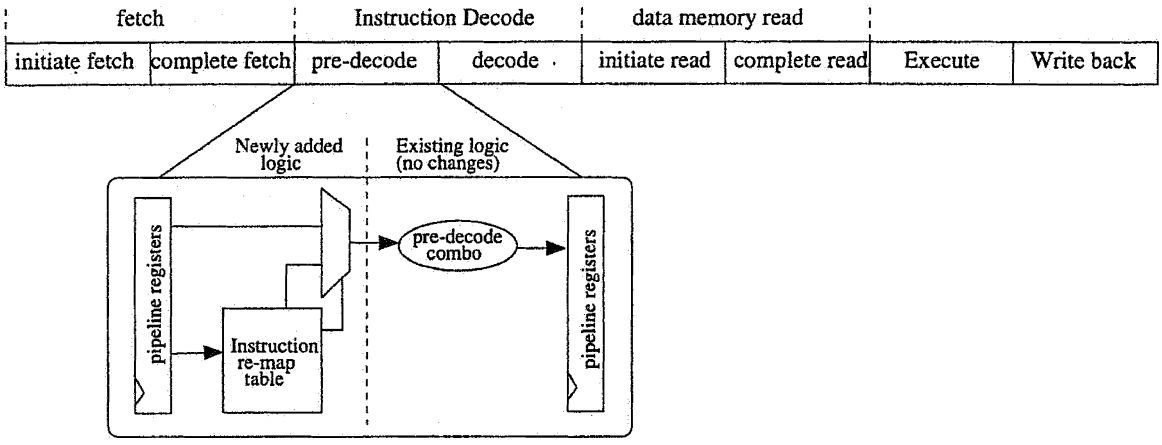

Figure 2. Modifications to CPU due to instruction re-map table.

stands for "Begin CONFiguration'' and *ECONF* for "End CONFiguration.'' This pair of instructions is similar to the *call* and *return* instruction pair. The semantics of these instructions is described below.

BCONF <*conf. address*>: This instruction is decoded by the configuration controller in the IRT and a *CALL* instruction is passed on to the instruction decoder. In addition, a configuration bit in the re-map table is set. Then the instructions are fetched from the location $< conf. address >$ in the program memory. The configuration bit in the instruction re-map table, when set, shuts off the instruction decoder and enables the write controller of the re-map table. The write controller generates write address for the instruction re-map table starting from the first location. So a set of consecutive locations are written with the instructions that are fetched from $< conf. address >$. This configuration happens till an *ECONF* instruction is fetched.

ECONF : This instruction is decoded by the configuration controller. This instruction resets the configuration bit in the instruction re-map table. This turns on the instruction decoder. The configuration controller sends a *return* instruction to the instruction decoder and the program control transfers back to the calling program and starts executing the instructions.

A typical sequence in the program memory after compression will appear as shown in Fig. 3. First, a set of instructions I1, I2, and I3 are written into the instruction re-map table. These instructions in the static sequence of the original code are replaced by the respective compressed representations CI1, CI2, and CI3. In other words, the instructions I1, I2, and I3 in the program memory are now compressed. For static reconfiguration, this configuration sequence is inserted once in the beginning of the program. For dynamic reconfiguration, different configuration sequences to exploit the instruction usage in different code segments are inserted in the beginning of each segment of code. In the example shown in Fig. 3, the instruction re-map table is configured twice, once for code segment 1 and once for code segment 2.

It can be seen that our proposed implementation of configuring the re-map table using *BCONF* and *ECONF* is both versatile and efficient. Note that the *BCONF* instruction enables writing the uncompressed instructions on successive locations in the

code segment 1		
BCONF <caddr1>	caddr1:	I1
CI1		I2
CI2		I3
CI3		ECONF
I4		
CI1		
...		
code segment 2		
BCONF <caddr2>	caddr2 :	I8
CI8		I9
CI9		...
CI8		ECONF
I6		
...		

CIm denotes compressed representation of instruction Im.

Figure 3. Program after compression.

re-map table starting from $Loc1$, until an *ECONF* instruction is encountered. Thus, it is possible to only re-write a part of the re-map table during each configuration leaving the rest of the re-map table unchanged. Thus it is possible to have a set $S1$ of instructions which are configured statically, that is only once at beginning of the program execution, and another set $S2$ of dynamically re-mapped instructions which are configured for different code segments. The set $S1$ that is statically configured is written at higher addresses of the re-map table. The set $S2$ that changes at different code segments occupies the lower addresses and gets reconfigured multiple times during the execution of the program. Thus without any additional support or overhead, a combination of static and dynamic encodings can be achieved.

It is also possible to achieve dynamic reconfiguration using multiple re-map tables that can have different configurations, all being configured statically. Dynamic reconfiguration of instructions can be achieved by switching to different re-map tables during different phases in running an application program. This method will eliminate the execution time overhead due to reconfigurations.

4.3.1. Care-abouts in Dynamic Reconfiguration A few important care-abouts in dynamic reconfiguration are discussed here.

First, using compressed instructions beyond branch target requires special care to be taken.

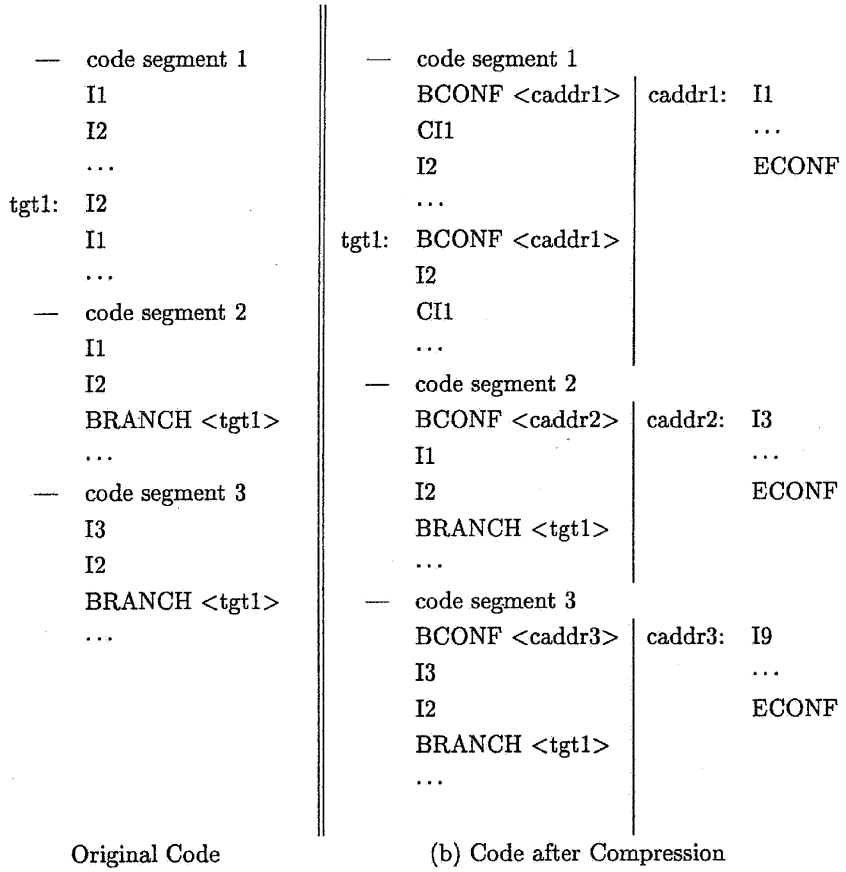

Figure 4. Branching across code segments.

This is illustrated with the help of an example. Figure 4 shows the original code where *tgt*1 was reachable by a branch instruction from code segment 2 and code segment 3 that may have different configurations of instruction re-map table. Hence an additional configuration instruction must be inserted at the branch target *tgt*1 to reconfigure the instruction re-map table with the configuration sequence meant for code segment 1. This is in addition to the configuration instruction at the beginning of code segment 1. This results in incurring the configuration cost twice if *tgt*1 is reached from code segment 1. If *tgt*1 is reached more often from code segment 1 than from code segments 2 or 3, then further optimization to avoid re-execution of *BCONF* instruction at *tgt*1 can be achieved by introducing an additional branch instruction.

Second, a subroutine may be called from different parts of the code. Therefore, upon entering into the subroutine, the instruction re-map table has different configurations at different points of time depending on the code segment from which it branched into the subroutine. Therefore, either the subroutine has to be left uncompressed or a separate configuration instruction is added to the subroutine to configure the instruction re-map table to exploit instruction usage inside it. If the subroutine configures the IRT, then upon returning to the calling program the IRT has to be reconfigured. Since the overhead associated in configuring the instruction re-map table twice for every call to a subroutine is higher, we leave the subroutines uncompressed in our analysis.

Last, branch instructions themselves are not compressed. Further, the branch target addresses have to be re-adjusted after the replacement of original instruction with compressed instructions, since the relative locations of the instructions are now altered. However this is not an issue as we assume that the instructions are assembled after re-map instructions are introduced.

5. Code Size Reduction

In this section, we report the code size reduction achieved by static and dynamic reconfigurations. We use two real life embedded control application programs[1] to evaluate the improvement in code size. These benchmarks are named BM1 and BM2. We use Texas Instruments TMS320c27x DSP as the base case. The instruction set of TMS320c27x DSP core is optimized for embedded control applications. We use the code generated for TMS320c27x DSP core for these benchmark programs. We compare the results of our approach with the original TMS320c27x code for the benchmark programs.

5.1. Static Reconfiguration

In this scheme, we encode the instructions once, prior to running a given application with the objective of reducing the code size for that particular application. Different applications can have different encodings for the same set of instructions but within a given application the encodings for all the instructions are fixed. With static reconfiguration, each application has fine tuned encodings for the instructions.

In this method, each of the application program is individually profiled to get instruction usage information. Each unique instruction is assigned a number called *InstructionUsageIndex*. *InstructionUsageIndex* of an instruction is a measure of the number of times that particular instruction is present in the entire static code. Higher the number of times an instruction appears in the static code, higher is the value of its *InstructionUsageIndex*. These unique instructions are then sorted based on the values of *InstructionUsageIndex* and the top m instructions are identified for compression, where m is equal to the number of locations in the IRT. The details of mapping instructions to smaller length encodings are detailed in Subsection 4.3. After the mapping, m most commonly used instructions have smaller width for their binary representation and hence occupy less space in program memory. In other words, now the code size has reduced.

In computing the code size for the compressed code, we take into account the configuration overhead (re-map table size and the instructions added for configuring the re-map table) and the overhead due to the additional bit for each uncompressed instruction (refer to Subsection 4.1 for details).

First, we allow flexible width for the compressed instructions. By this we mean that the compressed encodings can have any width, i.e., $2, 3, 4, \ldots, 12$ bits corresponding to the instruction re-map table of size $2, 4, 8, \ldots, 2048$. We report the code size reduction with flexible width for compressed instructions. Subsequently, we study the code size reduction when the compressed instruction width is restricted to 8 bits, for practical reasons, for re-map table of sizes from 16 to 128.

5.1.1. Flexible Size for Compressed Instructions We study the code size improvements for table sizes 4 to 2,048. Table sizes beyond 2,048 are not analyzed as the improvements achieved are low.[2] The width of the compressed instruction is given by $log_2(m) + 1$, where m is the number of locations in the instruction re-map table. Recall that the additional one bit is for identifying whether or not an instruction is compressed. The percentage code size reduction is computed using Eq. 1.

$$\%\ codeSizeReduction = \frac{OrginalCodeSize - CompressedCodeSize}{OrginalCodeSize} * 100 \quad (1)$$

In all our experiments, the original code considered is a highly optimized code. The code size reduction

Table 1. Code size reduction in static reconfiguration with flexible size for compressed instructions.

Table size (number of locations)	Compressed instruction size (bits)	Percentage code size reduction	
		BM1	BM2
4	3	−2.74	−3.19
8	4	−0.67	−1.76
16	5	2.01	0.22
32	6	4.42	2.06
64	7	6.66	4.16
128	8	8.86	5.82
256	9	10.73	6.96
512	10	11.66	6.92
1,024	11	10.88	5.07
2,048	12	6.98	0.02

for static reconfiguration with flexible width for compressed instructions is summarized in Table 1.

For both the benchmark programs, tables sizes of 4 and 8 result in negative impact on code size. This is because, in our approach, even the uncompressed instructions incur a 1 bit overhead. For small table sizes, vast majority of instructions are uncompressed. Therefore, for small table sizes (less than or equal to eight entries) the overhead incurred by the uncompressed instructions is higher than the improvement achieved by compressing a small number of instructions. This results in an overall increase in code size. For table sizes greater than or equal to 16 locations, we observe that the improvement due to compressed instructions is higher than the overhead incurred, thereby resulting in a considerable improvement in code size. The highest improvement is obtained for a table size of 512 entries for both the benchmarks. A code size reduction of 11.7 and 7.0% were observed for the two benchmarks. Table sizes beyond 512 locations show a decline in code size improvement. This is because of the following reasons: (i) overhead due to larger table is higher; (ii) the instruction usage index is high only for the top few hundred instructions leading to diminishing returns for higher table sizes; and (iii) the width of the compressed instruction increases thereby resulting in a decrease in the improvement. Therefore re-map tables of sizes greater than 1,024 entries show a sharp decline in the overall improvement.

5.1.2. Restricted Size for Compressed Instructions In the results presented in Table 1, we have calculated code size reduction by assuming the compressed instruction size to be $1 + log_2(m)$. To make the hardware implementation simple, uniform, and more practical, it may be required to restrict the compressed instruction size to 8 bits for all the table sizes less than or equal to 128. Table 2 summarizes the code size reduction where the size of the compressed instruction is 8 bits irrespective of the table size. With the 8-bit restriction, out of which 1 bit is used to indicate compressed instruction, we have only 2^7 possibilities for compressed instructions. So we restrict maximum table size to 128 entries in our evaluations. In this study, we observe a slight decline in the code size reduction compared to the results presented in Table 1. Despite this, we note that the static reconfiguration results in a code size reduction of 8.9 and 5.8% in these two benchmarks.

Table 2. Code size reduction in static reconfiguration with restricted size for compressed instructions.

Table size (number of locations)	Compressed instruction size (bits)	Percentage code size reduction	
		BM1	BM2
16	8	−0.25	−1.56
32	8	2.26	0.37
64	8	5.20	2.97
128	8	8.86	5.82

5.2. Dynamic Reconfiguration

In this scheme, the instructions can be re-mapped to smaller encodings on the fly, possibly many times, while running a given application program. The application program is first divided into several segments. Each segment is individually profiled to get the *InstructionUsageIndex* and the top m most commonly used instructions in that segment are compressed. In other words, each code segment can now be viewed as being configured individually. The code size after compression is obtained by adding the compressed code size of each of the code segment. The code size for a segment is computed in a manner similar to that in static reconfiguration, taking into account configuration overhead (for each code segment) and the single bit overhead indicating compressed/uncompressed instructions.

The static code sequence is first broken into several segments. We use a simple scheme for partitioning the code into several equal-sized segments. The code size reduction is computed for several such partitions and the partition that results in the smallest code size is chosen. Using additional heuristics for partitioning schemes may yield higher benefits. Each configuration is facilitated by inserting a configuration instruction—*BCONF* < *confRoutineAddr* >. The instructions that are to be written into the instruction re-map table are placed starting from the address < *confRoutineAddr* > in the program memory. The end of configuration is indicated by the instruction *ECONF*. Each configuration sequence can be viewed as a subroutine, hereafter called as *configuration routine*. Each segment has a configuration routine associated with it as illustrated in Fig. 3. Each segment now starts with a *BCONF* in-

Table 3. Code size reduction in BM1 for dynamic reconfiguration with flexible size for compressed instructions.

Table size (number of locations)	Compressed instruction size (bits)	Number of reconfigurations	Number of unique instructions compressed	Percentage code size reduction
4	3	91	161	0.12
8	4	92	321	2.99
16	5	91	593	5.72
32	6	43	590	8.07
64	7	21	591	9.66
128	8	9	608	11.11
256	9	2	394	11.30
512	10	1	512	11.66
1,024	11	1	1,024	10.88
2,048	12	1	2,048	6.98

struction that executes the appropriate configuration routine. This way, the instruction re-map table gets configured to suit the instruction usage of individual segments.

5.2.1. Dynamic Reconfiguration: Flexible Size for Compressed Instructions First, we analyze the code size improvement with flexible size for the compressed instructions. In computing the reduction in code size, we have taken into account the reconfiguration overhead and the per bit overhead for uncompressed instructions. The results for the two benchmark programs *BM*1 and *BM*2 are summarized in Tables 3 and 4.

In column 3, we report the number of reconfigurations (static count) required for different table sizes. In column 4, we report the number of unique instructions compressed in the entire program. It can be seen that the number of unique instructions compressed is much higher than the table size. This indicates that within a given application the instruction usage varies significantly across different code segments. Since we configure only once for static reconfiguration, higher table sizes are needed to cover a good portion of commonly used instructions. In dynamic reconfiguration, since the instructions are configured many number of times, the number of unique instructions compressed are high and this results in good improvements even for smaller table sizes. We observe that even with a small table size of 128 entries the gains are over 10% in both the applications, for dynamic reconfiguration.

Table 4. Code size reduction in BM2 for dynamic reconfiguration with flexible size for compressed instructions.

Table size (number of locations)	Compressed instruction size (bits)	Number of reconfigurations	Number of unique instructions compressed	Percentage code size reduction
4	3	98	202	1.61
8	4	84	361	4.49
16	5	57	534	7.36
32	6	29	606	9.43
64	7	20	872	10.21
128	8	10	969	10.08
256	9	5	1,083	9.06
512	10	2	963	7.49
1,024	11	1	1,024	5.07
2,048	12	1	2,048	0.02

Table 5. Code size reduction in BM1 for dynamic reconfiguration with restricted size for compressed instructions.

Table size (number of locations)	Compressed instruction size (bits)	Number of reconfigurations	Number of unique instructions compressed	Percentage code size reduction
16	8	91	593	0.86
32	8	43	590	4.19
64	8	21	591	7.36
128	8	9	608	11.11

5.2.2. Dynamic Reconfiguration: Restricted Size for Compressed Instructions As before, we evaluate code size reduction in dynamic compression by restricting the size of the compressed instructions to 8 bits for all table sizes up to 128 locations. Re-map table of size up to 128 locations can use 8-bit wide compressed codes. The code size improvement with restriction on the size for compressed instructions is summarized in Tables 5 and 6.

With restricted size for compressed instructions, dynamic reconfiguration gives much better results compared to static reconfiguration. A table size of 128 locations gives best results for dynamic reconfiguration in terms of achieving good code size improvement.

5.3. Partly Compressed Instructions

In our experiments thus far we have only considered compressing the entire instruction, i.e., including the source and the destination operands, into smaller width instructions. This means that a frequently occurring instruction is remapped to a smaller width opcode only if it has the same set of source and destination operands. The reason for doing this is that (i) it enables assigning compact (short) codes and (ii) although the frequency of repetition of an entire instruction (with same set of source and destination operands) is less than that of a part of the instruction, e.g., same opcode and source operand or same opcode and destination operand, there are still many opportunities to exploit the former. Our experimental results provide ample evidence for these arguments. Nonetheless, we experiment the impact of compressing only a part of the instruction, leaving either the source operand or the destination operand uncompressed (or separately encoded).

In this section we report the impact of partly compressed instructions on code size for both static and dynamic reconfigurations. We first choose four frequently used operand specifiers and retain their original encoding (uncompressed). We reserve 2 bits to represent uncompressed source or destination operands. Apart from the bit that is needed to indicate compressed instruction, we have only 5 bits with which we can have only 2^5 different compressed instructions. So we restrict the maximum table size to 32 locations in these experiments.

The results of static and dynamic reconfigurations are summarized in Tables 7 and 8. We observe that for a small table size (less than 32 entries), leaving source or destination operands uncompressed gives better results than the fully compressed instructions for the same table size. Also, we observe that leaving source as uncompressed gives relatively better results than leaving destination as uncompressed. This is because the number of different source

Table 6. Code size reduction in BM2 for dynamic reconfiguration with restricted size for compressed instructions.

Table size (number of locations)	Compressed instruction size (bits)	Number of reconfigurations	Number of unique instructions compressed	Percentage code size reduction
16	8	57	534	2.17
32	8	29	606	5.29
64	8	20	872	7.64
128	8	10	969	10.08

Table 7. Code size reduction in BM1 for partly compressed instructions.

Table size (number of locations)	Compressed instruction size (bits)	Percentage code size reduction			
		Source is uncompressed		Destination is uncompressed	
		Static	Dynamic	Static	Dynamic
8	8	−0.90	0.02	−1.40	−0.05
16	8	1.56	2.50	0.06	1.70
32	8	3.62	5.32	2.53	4.45

operands used in the code in general is larger than the number of different destination operands.

5.4. Access Time of Re-map Table

We evaluate the access time overheads introduced by the Re-map table using the CACTI simulator [39]. CACTI is a cache access cycle time model which is an analytical model based on transistor and wire characteristics derived from SPICE simulations. We use the CACTI simulator to obtain quickly the access time estimates for the Re-map table. The critical path for the Re-map table access is identical to any SRAM structure and primarily consists of decode, wordline, bitline and the sense amp stages. Hence, we measured the latency of the Re-map table[3] using CACTI 3.2. The access time for various Re-map table sizes and feature sizes are given in Table 9.

We do not include tag comparison time in the above estimate. However, the above access time estimates assume that each entry is of size 8 bytes, although in reality, each Re-map table entry is only 2 bytes long. This will marginally reduce the above access time values. As can be seen from the table, for a 0.13μ technology, the access time of a 128-entry Re-map table is less than 0.5 ns. We note that the microcontrollers studied in this paper, for the above feature size, typically operate at a clock frequency of 250 to 600 MHz, which correspond to a cycle time of 4 to 1.67 ns. If the microcontrollers operate at higher clock rates, then with 0.09μ feature size, a Re-map table access time of 0.4 ns can be achieved. We note that these values are relatively low compared to the decode time of embedded processors/microcontrollers. Hence we expect that the Re-map table access time can easily be included in the pre-decode stage without incurring any additional penalty.

6. Instruction Remapping for Power Reduction

In this section, we detail our approach to reduce power consumption using reconfigurable instruction encoding.

6.1. Motivation

In a processor, instruction fetching contributes to a significant portion of the overall power consumption. The primary contributor to the instruction fetch power is the toggling of high capacitance nets that connect the CPU and the memory. This power depends on the switching activity on these nets. This switching activity can be divided into two components, namely the switching activity on the program

Table 8. Code size reduction in BM2 for partly compressed instructions.

Table size (number of locations)	Compressed instruction size (bits)	Percentage code size reduction			
		Source is uncompressed		Destination is uncompressed	
		Static	Dynamic	Static	Dynamic
8	8	−2.18	0.20	−2.27	0.02
16	8	−1.37	2.88	−1.56	2.73
32	8	0.56	5.55	0.43	5.45

Table 9. Access time for re-map table.

Number of entries in Re-map table	Feature size		
	0.18μ	0.13μ	0.09μ
	(Access time in nanoseconds)		
32	0.61	0.44	0.30
64	0.68	0.49	0.34
128	0.69	0.50	0.35
256	0.76	0.55	0.38

address bus and that on the program data bus. The toggles in the program address bus are due to the changes in PC values used in fetching the instructions, while the toggles in the program data bus are due to the instructions fetched.

The toggles in the program data bus are likely to be higher and contribute significantly to the instruction fetch power. To establish this, we measure the toggles in program address and data buses in three proprietary benchmarks. These three benchmarks, referred to as BM-A, BM-B, and BM-C, are different from that used for analyzing reduction in code size. The reason for this is as follows. The benchmarks BM1 and BM2 used in evaluating code size reduction (in Section 5) are real life embedded control application programs. These are ideal for measuring code size reduction. However, many of the subroutines in these benchmarks (BM1 and BM2) will get executed based on real time interrupts. Power consumption depends on the dynamic trace. Hence, in the absence of interrupts, it may not be possible to measure the true power consumption. Benchmarks BM-A, BM-B, and BM-C are proprietary benchmarks designed to model real life scenario for measuring power consumption. Hence the use of these benchmarks meant for measuring power consumption is more realistic for analyzing the reduction in power consumption.

Table 10. Toggle distribution on program address and program data buses.

Benchmark	Percentage contribution to total number of toggles	
	Program address bus	Program data bus
BM-A	20.5	79.5
BM-B	15.9	84.1
BM-C	18.7	81.3

The distribution of toggles on program address and program data buses for these benchmarks is reported in Table 10. From Table 10, it can be seen that program data bus contributes to over 80% of the total number of toggles. The switching activity on program address bus is less since the instructions are fetched from consecutive locations till a discontinuity in instruction fetch due to a branch is encountered. However, there is no such relation for the values on program data bus which correspond to the instructions fetched. Since program data bus contributes to over 80% of the toggles, in this work, we focus on reducing the number of toggles on program data bus.

The number of toggles on program data bus depends on (i) the choice of encodings for the consecutive instructions in the dynamic sequence and (ii) the total number of fetches made. With reconfigurable instructions, we can reduce both the number of bytes fetched and the number of toggles between consecutive fetches of the compressed instructions in the dynamic sequence. That is, first, we reduce the number of bits fetched by compressing a set of most commonly occurring instructions in the dynamic sequence. Then, using Gray codes for encoding the instructions, we reduce the number of toggles between consecutive fetches of compressed instructions.

6.2. *Remapping Schemes*

As in the case of code size reduction, instruction remapping for power reduction can either be *static*, where a set of instructions are remapped to a set of encodings once at the beginning of the program, or *dynamic*, where different sets of encodings are achieved by doing the remapping multiple times while running an application program. While these static and dynamic reconfiguration schemes only remap instructions to reduce code size, they indirectly lead to reduction in instruction fetch energy due to the reduced number of bits fetched. First we study a naive assignment of compressed instructions to re-map table location. We refer to this scheme as *simple re-map scheme*. Subsequently, intelligent remapping of frequently occurring sequence of instructions to Gray codes is proposed which can result in further reduction in instruction fetch power. We refer to the later scheme as *Gray code re-map scheme*.

6.2.1. Static Reconfiguration To reduce energy spent on program data bus, we use a similar instruction re-map scheme that was described in previous sections on code density improvement. However, unlike in code size reduction where frequently occurring instructions in the *static code* or program is considered for remapping, in instruction remap scheme for energy reduction, the frequency of instructions in the dynamic instruction sequence is used to determine which instructions gets remapped with shorter opcodes. Thus, the entire dynamic instruction sequence is profiled to get *InstructionUsageIndex*. Based on this information, a set of top m most commonly used instructions in the dynamic sequence are chosen and are compressed. Due to this, the number of bits fetched reduces. The number of toggles is determined using a simple script that computes the sum of toggles between adjacent instructions fetched for the entire sequence. In calculating the number of toggles, the remapped smaller width encodings are used for remapped instructions.

6.2.2. Dynamic Reconfiguration As mentioned earlier, static reconfiguration does not exploit the varying usage of instructions within the dynamic sequence. In case of dynamic reconfiguration, we partition the entire dynamic sequence into several segments. We use a simple scheme that partitions the dynamic sequence into several equal sized segments. Each segment is profiled individually to get *InstructionUsageIndex*. Based on this, the most commonly used instructions are configured to have smaller encodings for each segment. The number of toggles for the entire dynamic instruction sequence is calculated with the remapped encoding. Several partitions are explored and the number of toggles is computed for each such partition. The partition that resulted in the lowest number of toggles is chosen. Better partitioning schemes that use additional heuristics may yield higher benefits.

In the above scheme, the number of partitions determines the number of times the instruction remap table is reconfigured. In this dynamic reconfiguration scheme all the careabouts listed in Section 4.3.1 must be taken care of.

6.2.3. Gray Coded Values for Compressed Instructions The simple re-map scheme discussed in the previous section arbitrarily assigns encodings for compressed instructions based on location in re-map table. However, a careful encoding for compressed instructions, based on Gray code, can further reduce the number of toggles. In this scheme, we encode the compressed instructions that are fetched in consecutive accesses using Gray coded values to further reduce the number of toggles. We analyze the sequence of instructions that appear most number of times in the dynamic trace. This sequence is then used as a base in determining the values for the encodings of compressed instructions. Since we do 16-bit fetches, and the compressed instruction is 8 bits, assignment of Gray coded values should be done in such a manner that it reduces the numbers toggles between consecutive fetches, rather than between adjacent instructions. We illustrate this below with an example.

Consider the following sequence.

$$I1$$
$$I2$$
$$I3$$
$$\ldots$$
$$Ik$$

Let $I1$ to Im form most commonly used sequence. Let these instructions be of size 16 bits in the original code. Let the compressed codes be of size 8 bits. Now after compression the instruction fetches will be as follows.

$$CI1 : CI2 \quad -> total\ of\ 16\ bits \quad (cycle1)$$
$$CI3 : CI4 \quad (cycle2)$$
$$\ldots$$
$$CIk-1 : CIk$$

Now we need to reduce the number of toggles between adjacent fetches and not between the instructions. This means we need to minimize the number of toggles between CI1:CI2 and CI3:CI4 and between CI3:CI4 and CI5:CI6 and so on[4] with the constraint that encodings of CI1, CI2, CI3, CI4, ... should be different. So we assign the values as follows. If the same instruction that is already compressed appears in the sequence again then we have to retain the already assigned code.

$$I1 <= Graycode(0);$$
$$I2 <= Graycode(\tfrac{m}{2});$$
$$I3 <= Graycode(1);$$
$$I4 <= Graycode(\tfrac{m}{2}+1);$$
$$\ldots$$
$$\ldots$$

where m is the number of locations in the table. This assignment will reduce the number of toggles

Table 11. Reduction in switching activity for simple re-map scheme.

Table Size (number of locations)	Percentage reduction in number of toggles					
	Static			Dynamic		
	BM-A	BM-B	BM-C	BM-A	BM-B	BM-C
4	−5.42	30.68	10.10	5.60	39.19	29.09
8	2.07	36.51	27.77	20.81	46.49	43.78
16	25.04	43.33	38.12	48.41	56.66	56.27
32	38.80	47.43	55.71	65.18	65.09	70.06
64	49.44	53.58	64.97	60.18	68.98	77.30
128	52.63	65.30	64.16	64.45	70.32	64.16

between adjacent fetches thereby reducing total number of toggles. Note that the Gray coding scheme can be combined either with static or with dynamic remapping.

6.3. Toggle Reduction in Remapping Schemes

In this subsection we study the toggle reduction for the three proprietary benchmarks BM-A, BM-B, and BM-C under various remapping schemes discussed in Section 6.2. We use different sets of benchmarks for analyzing reduction in code size and power consumption.

The percentage reduction in number of toggles is computed using Eq. 2.

$$\%\ toggleReduction = \frac{\#\ of\ toggles\ in\ original\ program. - \#\ of\ toggles\ after\ remapping.}{\#\ of\ toggles\ in\ original\ program} * 100 \quad (2)$$

The results for the percentage reduction in number of toggles for the simple re-map scheme are summarized in Table 11. We observe that for static reconfiguration, the improvement is higher with larger tables. Even with a table size of 128 locations we are able to get approximately 60% reduction in number of toggles. In case of dynamic reconfiguration, different instructions get compressed in different segments thereby exploiting the varying instruction usage within the dynamic sequence. The number of unique instructions that are compressed are therefore much higher than the table size. This is evident from the improved reduction in number of toggles for the dynamic reconfiguration scheme in Table 11.

With dynamic reconfiguration, we observe that even with small table sizes we can obtain much lower number of toggles compared to static reconfiguration. The improvements increase as the table size increases and then saturate. Results for benchmark BM-C

Table 12. Reduction in switching activity with gray codes.

Table Size (number of locations)	Percentage reduction in number of toggles					
	Static			Dynamic		
	BM-A	BM-B	BM-C	BM-A	BM-B	BM-C
4	−3.65	32.36	11.00	6.48	39.69	29.39
8	2.473	36.25	30.28	24.47	47.02	47.87
16	25.96	43.90	40.20	51.06	57.00	57.83
32	39.04	48.22	57.70	65.66	65.84	71.18
64	50.34	53.96	66.33	62.06	70.72	79.82
128	56.80	65.85	68.58	72.52	70.50	68.58

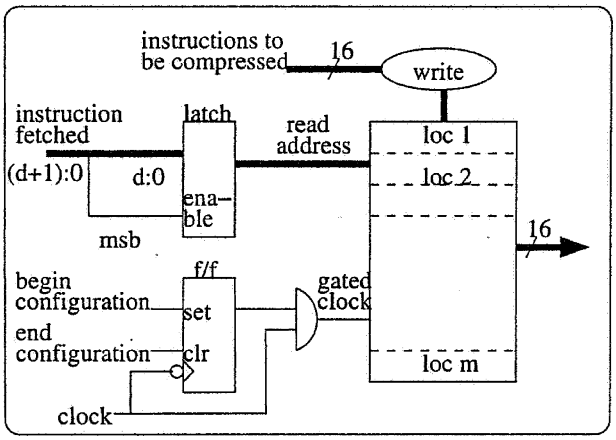

Figure 5. Power reduction in instruction re-map table.

indicate a dip in the improvement as the table size is increased to 128. This may be due to the increased reconfiguration overhead.

The impact of Gray coded values for compressed instructions on the number of toggles is summarized in Table 12. We observe that with Gray coded values for compressed instructions, the total number of toggles reduce further. For smaller table sizes, dynamic reconfiguration gives much better results compared to static reconfiguration and for larger tables sizes, the improvement over static reconfiguration is relatively less.

6.4. *Energy Efficient IRT Design and Evaluation*

In the previous section, we focused on reducing the number of toggles. To compute the actual savings in energy, we need to determine the energy saving due to reduced toggles. Further, the energy overhead incurred due to access to *Instruction Re-map Table* should be accounted for. The energy consumed by the *Instruction Re-map Table* is due to *writing into the table during configuration* and *reading of compressed instructions from the table*. The energy numbers reported in this paper take into account the energy consumed by the accesses (read as well as write) to the re-map table and its size.

The remap table is optimized for low power as shown in Fig. 5. Since the energy spent on clock network is high, we enable the clocks only during configuration of the table. The clocks are shut off once the table is configured. Similarly the read address is changed only when compressed instruction is fetched from the memory. In case of uncompressed instructions, the read address to the re-map table is not changed.

The clock to instruction re-map table is enabled only during reconfiguration. The *BCONF* instruction sets a bit that enables the clock to the instruction re-map table. *ECONF* instruction resets this bit and

Table 13. Configuration of instruction re-map table: variation of energy consumption with table size.

Table size (number of locations)	Normalized energy per write (write to one location)
4	1.00
8	2.17
16	4.16
32	8.06
64	12.44
128	21.81

Table 14. Reading from instruction re-map table: variation of energy consumption with table size.

Table size (number of locations)	Normalized energy per read (read from one location)
4	1.00
8	1.13
16	1.26
32	1.64
64	2.71
128	3.87

hence clock is disabled. So the clock network is made active only during configuration of the re-map table. This reduces the power consumed by clock network. Further to reduce the power consumed in unwanted reads, a latch is introduced for the address bits. We need to read from the instruction re-map table only when a compressed instruction is fetched. If the fetched instruction is in uncompressed form, then it is directly passed to the instruction decoder. We have added one bit to each instruction that indicates if it is a compressed representation. This bit that indicates whether it is a compressed instruction, is connected to the enable pin of the level sensitive latch. Therefore the latch passes the compressed instruction as read address to the re-map table only for compressed instructions. In case of uncompressed instructions the latch retains the old value thereby eliminating the energy spent in unwanted reads from the instruction re-map table. In other words, in case of uncompressed instructions, the reads are disabled.

Energy measurements are made as follows. We obtain the switching activity information on the CPU-memory interface and on the instruction re-map table using simulations (by fetching the compressed code that includes reconfiguration instructions from the memory). This switching activity is then back-annotated on the cells and nets that are part of CPU-memory interface and the instruction re-map table. We use Synopsys Power Compiler^{TM} [35] to obtain power numbers. The power numbers are then multiplied with the simulation time to get the energy numbers.

Writing to the re-map table takes much higher power as compared to reading from the same. The variation of energy consumption for write (configure) and read for various table sizes is shown in Tables 13 and 14.

As can be seen, as the table size increases the energy spent in writing to a location in the table increases significantly. The energy spent per read of a location also increases as the table size increases but the increase is much less compared to the increase in energy spent in writing.

Energy reduction for static and dynamic reconfigurations for different table sizes are summarized in Table 15. We compute the energy reduction for the configurations that resulted in best toggle rate. The number of times the table gets configured is also listed. Table size 0 corresponds to no configuration (original encoding for instructions). The normalized energy spent in three benchmarks for various table sizes and for static and dynamic reconfigurations are reported in columns 5–10. Columns 2–4 represent the number of times reconfiguration takes place during the execution of a benchmark. Note that this is a dynamic count, unlike the static count of the number of reconfigurations reported in Tables 5 and 6. We observe that smaller table sizes provide better energy reduction though the total number of toggles are more compared to bigger table sizes. This is because larger tables consume more energy during write and read operations as shown in Tables 13 and 14. The energy consumed is also a function of the number of times the table is reconfigured. We observe that for static reconfiguration, a table size of 32 locations is good enough to yield energy savings of over 40% for *BM-B* and *BM-C*, and 30% for *BM-A*. With larger table size, there is a decline in energy reduction. This is due to larger energy overhead for larger re-map table. With dynamic reconfiguration, we are able to achieve energy reduction of 47% for *BM-B* and *BM-C* and 35% for *BM-A*. Even a table size of 16 locations gives very good reduction in energy for dynamic reconfiguration. For *BM-C*, we observe that table size of 16 locations resulted in too many reconfigurations (149 times) to achieve good reduction in number of toggles. Due to this high number of reconfigurations, the energy overhead due to configuring the re-map table (writes) is high. Due to this, the improvement is somewhat low compared to the other benchmarks. For *BM-C*, table size of 64 gives the best energy savings though the number of toggles is not the lowest, due to lower reconfiguration overhead.

7. Conclusions

We proposed a mechanism *Instruction Re-Map Table* which acts as a decompression unit with minimal hardware overhead. We explained an incremental redesign of the TMS320c27x CPU to include the instruction re-map table. We showed that dynamic encoding helps in achieving higher benefits for both code size and power with smaller tables. Small tables are desirable to reduce the access time of the table so that the decompression latency can be hidden. We retain the memory interface speed since memory access times form the bottleneck in deep sub-micron designs. In our approach, with small table implemented as a

Table 15. Energy savings.

Table size (number of locations)	Number of reconfigurations (dynamic)			Energy spent (Normalized)					
				Static			Dynamic		
	BM-A	BM-B	BM-C	BM-A	BM-B	BM-C	BM-A	BM-B	BM-C
0	0	0	0	1	1	1	1	1	1
4	27	124	52	1.06	0.71	0.92	0.98	0.65	0.75
8	129	63	75	0.99	0.66	0.75	1.13	0.59	0.63
16	10	32	149	0.79	0.60	0.68	0.65	0.53	0.63
32	10	64	64	0.70	0.59	0.56	0.72	0.78	0.61
64	4	22	9	0.73	0.61	0.58	0.86	0.83	0.53
128	3	4	2	1.04	0.64	0.72	1.32	0.72	0.72

register file inside the CPU, the decompression delay is hidden. We also showed that the same hardware mechanism can be used to target power reduction. We used real life application programs as benchmarks in our analysis. In our evaluations, we have taken into account the overhead due to instruction re-map table. We observe that for code size reduction, larger tables give higher benefits while for power reduction, smaller tables give higher benefits. We showed that code size improvement of over 10% on an optimized code can be achieved and about 40% instruction fetch energy can be reduced. Further, our proposed scheme allows us to leave the time critical part of the code sequence (like real time interrupt service routines) uncompressed. Since no reconfiguration is required upon the entry or exit of the time critical routine, there is no performance penalty (due to additional cycles required for reconfiguration) for time critical code.

Notes

1. Due to proprietary nature of these applications, we do not disclose the names of these applications.
2. Further such re-map table sizes also incur a large access time. However, as shown in our experiments, a re-map table of moderate size, 256 or 512 entries, actually achieves the maximum performance for the considered benchmarks.
3. The Re-map table is similar to a direct mapped cache except that no tag compares are required.
4. Note that this does not necessarily require pairs of consecutive instructions, (e.g., I1 and I2) should both be compressible. If, say I2 is not compressible then CI1 and a part of I2 will be fetched in one cycle. Since encoding of I2 is fixed, the encoding of CI1 is determined such that the number of togglesis minimal.

References

1. Advance RISC Machines Ltd., "An Introduction to Thumb," March 1995.
2. L. Benini, G. Micheli, E. Macii, D. Sciuto, and C. Silvano, "Asymptotic Zero-Transition Activity Encoding for Address Busses in Low-Power Microprocessor-Based Systems," in *Proceedings of the 7th Great Lakes Symposium on VLSI*, pp. 77–82, Urbana-Champaigne, IL, March 1997.
3. L. Benini, F. Menichelli, and M. Olivieri, "A Class of Code Compression Schemes for Reducing Power Consumption in Embedded Microprocessor Systems," *IEEE Trans. Comput.*, vol. 53, no. 4, 2004, pp. 467–482.
4. A. Chandrakasan and R. Brodersen, "Low Power Digital CMOS Design," Kluwer, 1995.
5. K.D. Cooper and N. McIntosh, "Enhanced Code Compression for Embedded RISC Processors," in *Proceedings of the Conference on Programming Language Design and Implementation*, pp. 139–149, Atlanta, GA, May 1999.
6. S. Debray, W. Evan, R. Muth, and B. de Sutter, "Compiler Techniques for Code Compression," *ACM Trans. Program. Lang. Syst.*, 2000, pp. 378–415.
7. S. Devadas, S. Liao, and K. Keutzer, "Code Density Optimization for Embedded DSP Processors Using Data Compression Techniques," in *Proceedings of the 16th Conference on Advanced Research in VLSI*, pp. 272–285, Chappel Hill, NC, March 1995.
8. S. Devadas and S. Malik, "A Survey of Optimization Techniques Targeting Low Power VLSI Circuits," in *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, 1995.
9. D.A. Huffman, "A Method for the Construction of Minimum Redundancy Codes," *Proc. Inst. Electron. Radio Eng.*, pp. 1098–1101, vol. 40, September 1952.
10. M. Hiraki, R.S. Bajwa, et al., "Stage-Skip Pipeline: A Low Power Processor Architecture Using a Decoded Instruction Buffer," in *Proceedings of International Symposium on Low Power Electronics and Design*, Monterey, California, pp. 353–358, Monterey, CA, August 1996.

11. IBM, "*CodePack PowerPC Code Compression Utility User's Manual*," ver. 3.0, 1998. http://www-3.ibm.com/chips/techlib/techlib.nsf/products/CodePack.
12. T.M. Kemp, R.K. Montoye, J.D. Harper, J.D. Palmer, and D.J. Auerbach, "A Decompression Core for PowerPC," *IBM J. Res. Develop.*, vol. 42, no. 6, 1998. http://www.research.ibm.com/journal/rd/426/kemp.html.
13. K. Kissell, "MIPS16: High-density MIPS for the Embedded Market," Silicon Graphics MIPS Group, 1997. http://www.mips.com/Documentation/MIPS16whitepaper.pdf.
14. M. Kozuch and A. Wolfe, "Compression of Embedded System Programs," in *Proceedings of IEEE International Conference on Computer Design*, pp. 270–277, Cambridge, MA, October 1994.
15. J. Lau, S. Schoenmackers, T. Sherwood, and B. Calder, "Reducing Code Size with Echo Instructions," in *Proceedings of the International Conference on Compilers, Architectue, and Synthesis for Embedded Systems*, Oct. 2003.
16. L.H. Lee, W. Moyer, and J. Arends, "Instruction Fetch Energy Reduction Using Loop Caches For Embedded Applications with Small Tight Loops," in *Proceedings of International Symposium on Low Power Electronics and Design*, pp. 267–269, San Diego, CA, August 1999.
17. C. Lefurgy and T. Mudge, "Code Compression for DSP," Technical Report CSE-TR-380-98, *Presented at CASES-98 Workshop*, 1998.
18. C. Lefurgy, P. Bird, I-C. Chen, and T. Mudge, "Improving Code Density Using Compression Techniques," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 194–203, Research Triangle Park, NC, December 1997.
19. C.R. Lefurgy, "Efficient Execution of Compressed Programs," Ph.D. dissertion, University of Michigan, 2000.
20. H. Lekatsas, J. Henkal, and W. Wolf, "Code Compression for Low Power Embedded System Design," in *Proceedings of the 37th ACM/IEEE Design Automation Conference*, pp. 294–299, Los Angeles, CA, June 2000.
21. H. Lekatsas and W. Wolf, "Code Compression for Embedded Systems," in *Proceedings of the 35th ACM/IEEE Design Automation Conference*, pp. 516–521, San Francisco, CA June 1998.
22. H. Lekatsas, W. Wolf, and J. Henkel, "Arithmetic Coding for Low Power Embedded System Design," in *Proceedings of Data Compression Conference*, pp. 430–439, Snowbird, UT, March 2000.
23. C.H. Lin, Y. Xie, and W. Wolf, "LZW-based Code Compression for VLIW Embedded Systems," in *Proceedings of the Design, Automation and Test in Europe (DATE'04)*, pp. 76–81, Paris, France, February 2004.
24. S. Manne, A. Klauser, and D. Grunwald, "Pipeline Gating: Speculation Control for Energy Reduction," in *Proceedings of International Symposium on Computer Architecture*, pp. 132–141, Barcelona, Spain, June 1998.
25. M. Mehendale, S.D. Sherlekar, and G. Venkatesh, "Extensions to Programmable DSP Architectures for Reduced Power Dissipation," in *Proceedings of International Conference on VLSI Design*, pp. 37–42, Chennai, India, January 1998.
26. S.K. Menon and P. Shankar, "An Instruction Set Architecture Based Code Compression Scheme for Embedded Processors," in *Proceedings DCC 2005 Data Compression Conference*, p. 470, Snowbird, UT, March 2005.
27. E.G. Nikolova, D.J. Mulvaney, V.A. Chouliaras, and J.L. Nunez-Yanez, "A Compression/Decompression Scheme for Embedded Systems Code," in *1st ESC Division mini-conference*, 2003.
28. R. Owens, H. Mehta, and M. Irwin, "Some Issues in Gray Code Addressing," in *Proceedings of the 6th Great Lakes Symposium on VLSI*, pp. 178–180, Ames, IA, March 1996.
29. O. Ozturk, H. Saputra, M. Kandemir, and I. Kolcu, "Access Pattern-based Code Compression for Memory-constrained Embedded Systems," in *Proceedings of the Design, Automation and Test in Europe (DATE'05)*, vol. 2, 2005, pp. 882–887.
30. C. Piguet, P. Volet, J.-M. Masgonty, F. Rampogna, and P. Marchal, "Code Memory Compression with Online Decompression," in *Proceedings of the 27th European Solid-State Circuits Conference*, pp. 389–392, Villach, Austria, September 2001.
31. J. Rabaey and M. Pedram,"Low Power Design Methodologies," Kluwer Academic. Publishers, 1996.
32. Siemens Incorp, "TriCore Architecture Manual," 1997.
33. M. Stan and W. Burleson, "Bus-Invert Coding for Low Power I/O," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 3, 1995, pp. 49–58.
34. C. Su, C. Tsui, and A. Despain, "Saving Power in the Control Path of Embedded Processors," *IEEE Des. Test Comput.*, vol. 11, 1994, pp. 24–30.
35. Synopsys Inc., "Power Compiler Reference Manual," ver.2000.11, 2000.
36. Texas Instruments, "TMS320C27x DSP CPU and Instruction Set Reference Guide," March 1998. http://www.ti.com/sc/docs/storage/dsp/tech_doc.htm.
37. J.L. Turley, "Thumb Squeezes ARM Code Size," *Microprocess Rep.*, vol. 9, no. 4, 1995.
38. C.-Y. Wang and K. Roy, "Control Unit Synthesis Targeting Low-Power Processors," in *Proceedings of IEEE International Conference on Computer Design*, pp. 454–459, Austin, TX, October 1995.
39. S.J.E. Wilton and N.P. Jouppi, "CACTI: An Enhanced Cache Access and Cycle Time Model," *IEEE J. Solid-State Circuits*, vol. 31, no. 5, May 1996, pp. 677–688.
40. A. Wolfe and A. Chanin, "Executing Compressed Programs on an Embedded RISC Architecture," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 81–91, Portland, OR, November 1992.

Subash Chandar Govindarajan received his B.E. (Electronics and communications) degree from Coimbatore Institute of Technology, Coimbatore in 1995. He received his M.Sc.[Engg] degree from Indian Institute of Science, Bangalore in 2002. He has been with Texas Instruments, Bangalore since 1995 and is currently a Senior Member of Technical Staff. His current research interests are in the areas of reconfigurable architectures and low power design. He has 2 patents and seven publications in refereed international conference proceedings.
subba@ti.com

Mahesh Mehendale leads the Centre of Excellence for System-on-a-Chip (SoC) design at Texas Instruments (India), where he focuses on the methodologies and design of SoCs targeted to multiple DSP markets. In recognition of his technical leadership, he was elected TI Fellow in 2003. Mahesh has done B. Tech (EE), M.Tech (CS&E) and Ph.D. all from the Indian Institute of Technology, Bombay. He has published more than 35 papers, presented many invited talks/tutorials and also participated on panel discussions at many international conferences. He also has co-authored a book on VLSI synthesis of DSP kernels published by Kluwer. Mahesh holds six US patents and was elected senior member of IEEE in 2000. His areas of interest include VLSI design methodologies and DSP/SoC architectures.
m-mehendale@ti.com

R. Govindarajan received his B.Sc. degree in Mathematics from Madras University in 1981 and B.E. (Electronics and Communication) and Ph.D. (Computer Science) degrees from the Indian Institute of Science, Bangalore in 1984 and 1989, respectively. He has held postdoctoral and visiting researcher positions at Canadian and US universities. He has been with the Supercomputer Education and Research Centre and the Department of Computer Science and Automation, Indian Institute of Science, Bangalore, since 1995.

His current research interests are in the areas of High Performance Computing, compilation techniques for instruction-level parallelism, compilation techniques for DSP and embedded processors, distributed shared memory architectures, and cluster computing. He has more than 80 publications in these areas in international journals and refereed international conference proceedings. He is a Senior Member of the IEEE and a member of the IEEE Computer Society.
govind@serc.iisc.ernet.in