Speeding Up Control-Dominated Applications through Microarchitectural Customizations in Embedded Processors*

Peter Petrov and Alex Orailoglu

Computer Science & Engineering Department
University of California, San Diego
(ppetrov,alex)@cs.ucsd.edu

ABSTRACT

We present a methodology for microarchitectural customization of embedded processors by exploiting application information, thus attaining the twin benefits of processor standardization and application-specific customization. Such powerful techniques enable increased application fragments to be placed on the processor, with no sacrifice in system requirements, thus reducing the custom hardware and the concomitant area requirements in SOCs. We illustrate these ideas through the branch resolution problem, known to impose severe performance degradation on control-dominated embedded applications. A low-cost late customizable hardware that uses application information to fold out a set of frequently executed branches is described. Experimental results show that for a representative set of control dominated applications a reduction in the range of 7%-22% in processor cycles can be achieved, thus extending the scope of low-cost embedded processors in complex co-designs for control intensive systems.

1. INTRODUCTION

Embedded processors constitute currently an attractive solution for various modern electronic applications. Typically such designs are implemented as a part of a system-on-a-chip (SOC) IC, a trend accelerated by ever increasing transistor density and die sizes. The inherent drawbacks of processor-based designs, typically stemming from the embedded redundancy of the processor computational model, impose significant challenges though in achieving overall system requirements, particularly performance and power.

Processor performance and cost considerations play a significant role in the hardware/software co-design of systems [1]. A fundamental issue is the partitioning problem of system functionality between hardware and software. While shifting as much as possible of the functionality onto software reduces system cost and time-to-market, performance and power consumption suffers, thus counterbalancing the aforementioned advantages of general-purpose processors; the end result typically is unnecessarily large custom hardware in the SOC. Yet processor components with all their attendant benefits of flexibility and high volumes would be natural candidates for further utilization, had they been guaranteed not to impose performance and power impediments. We propose in this paper a research focus aimed at defining a new class of embedded processor architectures, hard-coreable yet simultaneously customizable per application, thus enabling larger parts of the codesign to be placed in software with no adverse performance or power impact. In this direction and as a first step in exploring the possibility of such a scheme, we target the branch resolution unit, a factor of high importance in evolving control-dominated embedded applications.

In this paper, we show that processor customization is a powerful tool for boosting performance and reducing power consumption of processor cores. A customized embedded processor micro-architecture can utilize application properties and make informed decisions during execution. We propose a technique that communicates application specific properties to the micro-architecture of the processor core. The application-specific properties are identified during compile time and provided to the hardware, so that they can be exploited efficiently during program execution.

The paper focuses on the ramification of the aforementioned ideas on the branch resolution logic in order to alleviate the detrimental effect of conditional branches for pipelined architectures. Especially for control intensive applications which are part of a typical reactive system, high-cost general-purpose branch predictors behave poorly, resulting in low performance and significant amount of power wasted in executing instructions from the mispredicted path. Of course, utilizing no branch prediction at all in these circumstances has even direr consequences, resulting in worse performance and power degradation, thus eventually pushing the control dominated parts of the application off the processor and into custom hardware.

Instead, utilization of knowledge about branch predicates, branch addresses, and branch target instructions can be exploited efficiently by the customizable branch resolving logic. A significant number of branches can thus be *folded out* during the fetch stage as advance knowledge of the branch direction ensures branch replacement with certainty. At the same time a smaller, cost-effective variant of a general-purpose predictor can be used for the remaining branches. Three major benefits accrue from this technique.

- A significant improvement in performance is achieved. This is especially true for control intensive applications that typically exhibit low predictability of branches.
- The total number of instructions passing through the pipeline is reduced, as a branch instruction folded in the fetch stage proceeds no further in the pipeline and no mispredicted instructions are executed. Consequently, power consumption is decreased.
- Comparable branch prediction accuracies can be achieved at significantly lower area costs.

*This work is supported by NSF Grant 0082325; the work of the first author is additionally supported by an IBM Graduate Fellowship.

DAC 2001, June 18-22, 2001, Las Vegas, Nevada, USA.

The fundamental advantage of processor-centric implementations is the flexibility to alter system functionality. It is paramount consequently to preserve this processor characteristic even post customization. In order to utilize customization benefits not only for a single application, but rather for a class of applications, it is highly desirable to be able to re-customize the processor between application runs, and in any case certainly in a post-manufacturing fashion. We propose to accomplish these goals through microarchitectural reprogrammability, utilizing application and architecture characteristics. While traditional methods of reconfigurability, such as FPGAs, fundamentally reliant on logic level customization, can deliver flexibility, they frequently do so only at the expense of high area and performance cost.

2. RELATED WORK

A great deal of research work has been recently completed in designing branch predictors for alleviating the detrimental effect of conditional branches on pipelined architectures. Static prediction approaches use both profiling and compile time analysis techniques to statically predict the branch instructions [2]. Dynamic branch predictors use run time information to improve their predictions [3]. One of the fundamental ideas is to exploit correlation between branch outcomes [4]. Recently, an approach for conditional branch folding for embedded applications has been proposed [5]. It is based on the detection and resolution of a subset of short loop branches.

In the customization arena, various approaches have been researched, but none that target the microarchitecture, the primary focus of this paper. In [6] a strategy for customizing a VLIW architecture is described. The approach explores the design space of the architecture by changing the number of functional units, clusters, and the number and size of register files. A different set of customization approaches considers application-specific instruction set extensions [7]. The efforts here include the design of specialized instruction sets for multimedia and communications applications.

3. MOTIVATION

A typical general-purpose branch predictor contains large tables with various branch patterns corresponding to different branches and global/local branch execution histories. A large branch target buffer is needed to cache the branch target address to handle the case of a taken prediction. Reducing the sizes of these tables in order to scale down the predictor typically results in significant prediction accuracy degradation. Yet such large hardware structures lead to excessive power consumption which is rarely acceptable in the domain of embedded applications. Furthermore, even these complex general-purpose predictors fail to deliver high prediction accuracy in the case of control intensive applications.

Current branch predictors are presumed to work on the basis of statistical representations and inferences. Nevertheless, the underlying correlations captured by them are comprised fundamentally of amalgams of data relationships between various predicate defining assignments and conditional branches. For instance, the code in Figure 1 exhibits correlation for B4. Although typical branch predictors will capture this B4 correlation as a statistical inference, one can note that what is being captured is a straightforward data inference from the predicate defining branch, B1, to the branch being predicted, B4. At the same time, as no data inference exists (if we assume all variables are unrelated), the same example will exhibit no branch prediction correlation for branch B5. Of course, general-purpose machines possess neither the hardware architecture, nor have an intimate knowledge of the application to enable them to exploit such correlations on anything other than a statistical basis.

```
if (c1) then  // Branch B1
   c4 = 1;
   ...
end if;
if (c2) then  // Branch B2
   ...
   if (c3) then  // Branch B3
      ...
   end if;
end if;
if (c4!=0) then  // Branch B4
   ...
end if;
if (c5) then // Branch B5
```

Figure 1: Direct data correlation

While there does exist a correlation between B4 and B1, which can be captured by current statistical branch prediction methods, albeit inefficiently as the correlation will be clouded by the intervening branches, an interesting further observation is the effect of the resolution of B2 and of the embedded B3 branch. It will be observed that the consequent imbalance in the control flow will deteriorate the statistical inference of general-purpose predictors by changing the position of the relevant branches inside the global branch history. For example, if branch B2 in Figure 1 is taken, only then branch B3 will appear in the global branch history, thus changing the relative position of the information-carrying branch B1 vis-a-vis the predicted branch B4.

One further aspect that plays havoc with the statistical inference process of branch predictors is a possible reliance on input data. This, in its most salient form, impacts the prediction of the branch whose predicate relies directly or indirectly on the input data. A more intricate case will be the impact of this input dependence and therefore hard-to-predict branch on the prediction of subsequent branches in the case of distance variance due to nested branches between predicting and predictor branches.

```
lh     r2, 0(r4)
subu   r3, r2, r11
addu   r4, r4, 2
sra    r2, r3, 31
andi   r13,r2, 0x0008
bgez   r3, Label
```

Figure 2: Branch depending directly on input data

A code fragment[1] illustrating the first case is given in Figure 2. In this case the branch outcome depends directly on the value produced by the load instruction. The connection between the *subu* instruction, defining the argument of the branch condition r3, and the branch, remains unnoticed by a general-purpose predictor. The result is significant performance degradation due to the high misprediction rate and, even more importantly for a large number of embedded systems, wasting power in executing instructions from the wrong path. A more intricate extension of the same problem consists of a hard-to-predict, because of input data dependence, branch, creating difficulties for subsequent branches, especially if the outcome of the hard-to-predict branch defines a predicate of another branch, and the number of branches between the correlated branches varies depending on intermediate branch resolution.

By targeting in an application-specific manner branches with properties such as the ones described above, significant performance improvements can be achieved. Utilizing the statically available application information can lead to a low-cost, efficient solution for early branch resolution and consequently to branch folding. We examine the principles of the application-specific branch folding methodology in the next section.

[1] This assembly code is part of the ADPCM Encode benchmark [8] and was produced by gcc for the SimpleScalar toolset [9].

4. BRANCH RESOLUTION

An interesting observation that accrues from the examples in section 3 is that in the cases of direct data inference the prediction step can be completely superceded and a complete branch resolution with consequent branch folding affected. The branch instruction will be replaced in the pipeline by its target instruction, thus completely being eliminated from the instruction stream. To achieve such substantial benefits, not only does the direct data inference need to exist, but furthermore the value of the predicate defining variable needs to be resolved prior to fetching the branch in question. Of course, these benefits can be attained in an embedded application but not in a general-purpose one, unless one is willing to change the instruction set architecture in a general-purpose processor to accommodate the proposed approach.

Typically, branch folding schemes for general-purpose processors are applied to unconditional branches [10]. Such branch folding schemes utilize a table composed of replacement instructions and corresponding destination addresses. The branch folding scheme that we propose needs similarly to capture the instruction and the destination address, both statically obtainable from the application code. Nonetheless, in the case of conditional branches, even when a direct data inference exists, the predicate sense, unlike an unconditional branch which is always taken, is not immutable, although known dynamically *a priori*. Each branch folding entry needs to be extended to include predicate sense information, consequently. Handling the fall-through cases necessitates additionally capture of the subsequent instruction in the fall-through path. No additional destination address needs to be stored though, as this is a direct function of the current PC.

For embedded processors, which due to more stringent power consumption limitations lack the capability for multiple instruction issue and out-of-order execution, the time interval between the branch condition register definition and the branch fetch can be considerable. Therefore, for a large number of branches, the register value that determines the branch condition is computed well before the branch is fetched. The *Application-Specific Branch Resolution* (ASBR) methodology we propose exploits this separation in time between branch fetch and branch condition register definition to compute the branch condition early, thus resolving the branch direction before the branch is fetched. Yet, in order to be able to fold such a branch, additional information about the branch is needed as well. This information, consisting of the previously described branch target address and the target and fall-through instructions, is "pre-decoded" statically during compile time and provided to the branch resolution logic. When the branch is subsequently fetched, it is identified and replaced with its target instruction. The feasibility of early branch condition calculation and branch folding depends on the interval available before the condition register is computed and the branch fetched. While ASBR is not inherently related to compiler technology, certain optimization techniques can boost significantly the effectiveness of the approach. Instruction scheduling [11] and software pipelining [12] can both be efficiently utilized to extend the time interval between branch condition definition and branch fetch.

Fundamentally, the ASBR methodology can be divided into two phases. The first phase, shown in Figure 3, performs the *Early Condition Evaluation.*

```
Early_Condition_Evaluation:
 if (committing Ri)
   Update(Conditions(Ri));
```

Figure 3: Early condition evaluation

In this step, the branch direction is computed prior to fetching the branch. Every time a register is being committed, all possible conditions associated with this register are updated. In a typical RISC architecture this set of conditions corresponds to a few zero comparisons. Performing the branch condition evaluation before the branch is fetched eliminates the need for the branch instruction to read the condition register from the register file and perform the comparison.

The second phase, shown in Figure 4, corresponds to the actual branch folding mechanism. It is performed during the fetch stage of the branch instruction.

```
ASBR:
   if (Fetch(PC)==branch_type)
      if (PC in {BA}) then
         if (PredicateStorage(DI)==taken)
            PC=BranchTargetAddress+4;
            instr=BranchTargetInstruction;
         else
            PC=PC+8;
            instr=BranchFallthroughInstr;
         end if;
      end if;
   end if;
```

Figure 4: Application-specific branch resolution

If the fetched instruction is a branch and if the PC of this branch matches a *Branch Address* (BA) from a set of branches, a series of actions is undertaken. *PredicateStorage(DI)* signifies the branch predicate value by using a *Direction Index (DI)* to index a storage table with the pre-computed branch directions associated to the branch condition register. This storage is updated by the first phase, the *Early Condition Evaluation*.

If the branch is taken, then the PC is updated with the *Branch Target Address* (BTA) incremented by the size of one instruction word, thus positioning the PC to the instruction subsequent to the branch target. The instruction word, currently containing the branch itself, is overwritten with the *Branch Target Instruction* (BTI). In the case of a *not taken* branch, the PC is positioned to the instruction subsequent to the *fall-through*, while the branch is replaced with the op-code of the *Branch Fall-through Instruction* (BFI).

The BA, DI, BTA, BTI, and BFI constitute a set of statically available information. This information is obtained statically during compile time and provided to the embedded processor core during program code upload. The application-specific, customizable branch resolution logic exploits it by performing *early condition resolution* with subsequent *branch folding.* The content of these tables can be dynamically re-defined at system level at run-time in order to accommodate customization for different program modules, if the application contains several important parts.

An essential issue to be addressed is the handling of branches for which the condition register definition instructions lie on different control paths, thus resulting in varying time intervals between branch fetch and condition evaluation. This variance can arise due to differences in the control paths through which the branch is reached. If it further happens that some of the paths guarantee that the predicate value will be calculated prior to branch fetch while other paths do not, a validity issue ensues if such a branch is to be folded. Resolution of branches with such condition dependency necessitates a mechanism that indicates the current validity of each predicate value. The value is invalid, if the register associated with it is currently being produced by an instruction still in execution. The required tracking of register usage can be accomplished through a counter associated with each register. The counter is incremented whenever an instruction that produces register R is decoded; when register R is committed, the counter is decremented. A zero value on the counter guarantees the validity of the pre-computed predicates associated to this register.

5. OPTIMIZATIONS FOR ASBR

The proposed application-specific branch folding techniques require knowledge of the predicate-defining register prior to branch fetch. In the case of pipelined architectures the distance between the predicate defining register statement and the branch needs to exceed the pipeline depth. In this section we show how the range of applicability of the proposed ASBR method can be significantly enlarged by utilizing certain compiler optimization techniques for increasing the effective $distance$, and also by microarchitectural optimizations for minimizing the effective $threshold$ imposed by the pipeline.

5.1 Compiler support

The compiler capability to schedule the instruction that defines the registers involved in computing the branch condition is crucial. Fortunately, most modern compilers support instruction scheduling for data dependent instructions in order to avoid pipeline stalls. In this case, the branch must be considered as a data dependent instruction on the condition register producing instruction.

Software pipelining [12], a technique for increasing the instruction-level parallelism by overlapping independent parts of loop iterations, effectively inserts branch independent instructions between the branch and the last instruction that defines the branch condition. A typical program fragment is shown in the left part of Figure 5, while the right part of the figure presents the same code after aggressive software pipelining for scheduling the branch condition evaluation.

```
                          prologue code;
for(i=1;i++;i<1000)       for(i=2;i++;i<999
  load F0 from memory;      Rt=f(F0);
  R=f(F0);                  if (R) then
  if (R) then                 R=Rt;
    ...                       ...
  else                      else
    ...                       R=Rt;
  end if                      ...
end for                     end if
                            load F0 from memory;
                          end for;
                          epilogue code;
```

Figure 5: Software pipelining for ASBR

5.2 Processor architecture

Pipeline depth determines the number of time slots required between the branch and the last instruction that defines the branch condition register. Therefore, the *threshold* for a given architecture is determined as the number of states between instruction fetch and register commit stages. One can notice though that this threshold can be reduced by updating the predicate value right after the execution stage immediately upon actual production of the value, instead of during the register commit stage. Direct paths from the ALU units to the early predicate evaluation logic are needed to accomplish this. These paths resemble the traditional forwarding paths in the pipeline, except in that they transfer a recently computed value to the early predicate evaluation logic instead of to a processor functional unit.

Figure 2 has already illustrated an example code that demonstrates a situation suitable for early branch resolution. If we assume that the execution stage is the 3^{rd} pipeline stage, the forwarding path to the early predicate resolution logic makes the branch predicate available at the end of the 4^{th} stage, thus determining the *threshold* value to be 3. An even more aggressive approach can be utilized that computes the predicate value at the end of the execution stage of the pipeline, in the case that the execution stage is not time critical and the clock period not adversely affected. A *threshold* value of 2 could then be attained. For example, the branch in the code shown in Figure 2 is branch-foldable under either of these pipeline forwarding path augmentations, as they result in *threshold* values of 3 and 2, respectively; since three independent instructions exist between the branch and the instruction defining the predicate register, an unaugmented five stage basic pipeline will not be able to fold the branch shown.

6. BRANCH SELECTION FOR ASBR

The ASBR methodology is intended to target only a subset of the branches that are feasible therein to resolve. This selectivity is a crucial premise for the cost-effectiveness of the application-specific customization, an issue to which we return in the subsequent section.

All branches that satisfy the distance property discussed are candidates for folding, while branches that fail cannot partake, but may continue instead to use a general-purpose branch predictor. Yet a certain amount of selectivity in choosing the branches to be folded may still be appropriate as the cost of folding a branch exceeds that of predicting it and as foldable branches may show highly variant expected benefits. Getting the utmost advantage from the proposed technique requires a prioritization of the foldable branches based on their expected benefit. Frequently executed, hard-to-predict branches are especially propitious to resolve by using ASBR. There are three fundamental advantages for utilizing ASBR for this particular subset of conditional branches.

- Resolving the frequently executed, hard-to-predict branches and folding them out with their target instructions results in proportionately greater performance and power improvements. This is a direct consequence of Amdahl's law.
- Applying the ASBR methodology to a small subset of branches results in less costly and power consuming implementation. As described in Section 4, certain branch information needs to be provided to the microarchitecture, which implies a linear growth in hardware complexity per branch.
- Eliminating the hard-to-predict branches decreases the destructive aliasing into the prediction table greatly; hence better prediction can be achieved with a significantly less expensive general branch predictor.

Not only does the proposed technique improve branch resolution through folding a number of branches, but furthermore the removal of hard-to-predict branches from the prediction table can actually improve the accuracy of the complementary, original branch prediction method. The overall twin boost to branch resolution can significantly reduce the number of execution cycles for embedded applications requiring high performance and low power or alternatively can be used to drastically reduce area and still keep the original branch prediction rates by using a much more lightweight branch predictor instead. Both the performance and power improvements or the alternative reductions in area are verified through experimental data in Section 8.

7. IMPLEMENTATION

In order to be able to implement the application-specific branch resolution technique, the branch information obtained through static analysis of the application code needs to be encoded into the processor and utilized. The branch characteristics utilized in ASBR as described in Section 4 are BA (the branch address), DI (direction index), BTA (branch target address), BTI (branch target instruction), and BFI (branch fall-through instruction). The corresponding values need to be captured in the ASBR logic.

In order to preserve processor flexibility and in order to continue making the proposed technique applicable even to hard core processor designs, we propose a scheme targeting a microarchitecturally

	ADPCM Encode			ADPCM Decode			G.721 Encode			G.721 Decode		
	Cycles	CPI	Acc	Cycles	CPI	Acc	Cycles	CPI	Acc	Cycles	CPI	Acc
not taken	12,232,809	1.85	32%	10,818,933	1.96	31%	80,695,528	1.73	53%	80,418,120	1.83	53%
bimodal	9,354,462	1.41	69%	7,909,813	1.44	71%	62,130,909	1.33	91%	62,820,828	1.43	91%
gshare	8,454,179	1.28	82%	7,267,628	1.32	81%	62,317,531	1.33	91%	63,128,743	1.44	90%

Figure 6: Branch predictability of the benchmarks

	br0	br1	br2	br3	br4	br5	br6	br7	br8	br9	br10	br11	br12	br13	br14	br15
exec #	200,000	200,000	200,000	25,000	23,514	25,000	25,000	25,000	25,000	24,995	150,000	150,000	1,761,060	23,514	24,997	25,000
not taken	0.99	0.74	0.51	1.00	0.51	1.00	1.00	0.00	0.99	0.52	0.00	0.94	0.89	0.51	0.49	1.00
bimodal	0.99	0.70	0.51	1.00	0.50	1.00	1.00	1.00	0.99	0.51	1.00	0.96	0.88	0.50	0.50	1.00
gshare	0.99	0.81	0.52	0.99	0.61	0.96	0.95	0.97	0.99	0.91	0.99	0.96	0.86	0.50	0.93	0.99

Figure 7: Execution statistics for the set of branches selected for G.721

reprogrammable implementation. This scheme must allow easy re-customization of the processor after application modification. The branch information must be redefined and exploited by the processor in the same way as the program code. Conceptually, this implies enlargement of the communication link between the compiler and the embedded processor. The *branch information* is loaded into the processor core in a similar way as the program code.

A Branch Identification Table (BIT) is utilized to store the *branch information*.

The fields in each BIT entry store the following information:

PC The PC field stores the program counter (address) of the branch. This branch address is used for branch identification in the fetch stage of the processor.

inst1, inst2 These fields correspond to BTI and BFI and contain the target and fall through instructions respectively. They are used in replacing the folded branch.

BA The program counter (address) of the target instruction. The processor program counter is updated from this field.

DI A direction index that points to a structure with branch conditions associated with each register. This field incorporates the information about the branch condition.

The BIT is looked up with the program counter during the fetch stage. Upon a match of the PC field and the processor program counter, the *direction index* field is used in determining the branch direction. Since this look-up is performed in the fetch stage prior to instruction decoding, the existence of the PC field in BIT is the factor that determines that the instruction is a branch and that all needed branch information is available in the BIT entry.

The number of entries in the BIT determines the number of branches that can be handled by the approach at any time instance. Since only the most frequently executed branches within the important application loops are targeted, a small number of BIT entries would suffice.

The direction index in the BIT entry points to an entry in the Branch Direction Table (BDT). The BDT is composed of an entry per processor register; each entry contains a number of direction bits for the architecture-supported branch conditions and a validity counter as discussed in Section 4. Figure 8 depicts a BDT for an architecture that supports 4 general-purpose registers and branch instruction conditions of *not equal to zero*, and *less-than-or-equal to zero*. Every time a register value is produced from a functional unit, the direction (condition) bits of the register entry in the BDT are updated.

If the application contains more than one loop and the BIT cannot cover all the branches from these loops, a mechanism is needed to add new branch definitions to the BIT. As discussed earlier, the size of BIT is kept small for performance and power efficiency reasons. An

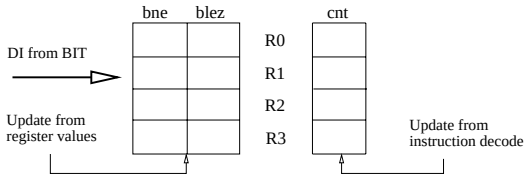

Figure 8: A four entry BDT example

effective way to virtually increase the size of BIT is to add additional copies of BITs and switch between them during the loop transitions. At any moment only one BIT copy will be active, thus not exceeding the power consumption or performance limitations. Activating a BIT copy can be performed by writing a special value to a control register just before entering the loop.

8. EXPERIMENTAL RESULTS

In our experimental framework we use architectural level simulations, thus measuring precisely the number of processor cycles needed to execute the benchmark. We use the most detailed simulator available in the SimpleScalar toolset [9], one capable of simulating a pipelined architecture. The architecture supports a MIPS-like instruction set with conditional branches supporting all possible zero comparisons. The fetch stage of the simulator is modified, so as to implement the approach described in this paper. A manual scheduling in the application code is performed for the branches that we identify as candidates for folding.

A pipelined architecture with a 5 stage pipeline, in-order single issue, is assumed for simulation. The memory hierarchy subsystem utilizes 8KB instruction cache, and 8KB data cache. This type of architecture corresponds closely to current embedded processor cores.

Four applications from the MediaBench collection of benchmarks [8] have been examined. The first two applications constitute the ADPCM (Adaptive Differential Pulse Code Modulation) encoder and decoder. The other two benchmarks used in our study are the G.721 (speech coding standard) encoder and decoder. We study the branch behavior and apply the proposed approach of application-specific, early branch resolution. As a comparison base, figure 6 reports execution results for all four benchmarks obtained by using well-known general-purpose branch predictors; total number of cycles, CPI, and accuracy measurements are given for each predictor. The general-purpose predictors examined for the baseline architecture are listed below:

not taken - Always predicts the branches as not taken. This is the default in many embedded processors that lack branch predictors;

bimodal [3]- a predictor that uses 2048 2-bit saturating counters and a branch target buffer containing 2048 entries;

gshare [3] - a 2 level global history correlation predictor utilizing an

	br0	br1	br2	br3
exec #	147,520	147,520	147,520	147,520
not taken	0.48	0.31	0.48	0.50
bimodal	0.43	0.63	0.43	0.50
gshare	0.61	0.65	0.84	0.91

Figure 9: Execution statistics for the set of branches selected for the ADPCM encode benchmark

	br0	br1	br2
exec #	147,520	147,520	147,520
not taken	0.50	0.31	0.48
bimodal	0.00	0.63	0.43
gshare	0.91	0.88	0.59

Figure 10: Execution statistics for the set of branches selected for the ADPCM decode benchmark

	ADPCM Encode		ADPCM Decode		G.721 Encode		G.721 Decode	
	Cycles	Impr.	Cycles	Impr.	Cycles	Impr.	Cycles	Impr.
not taken	10,328,867	**16%**	9,367,586	**13%**	76,089,314	**6%**	80,418,120	**5%**
bi-512	7,282,057	**22%**	6,321,949	**20%**	57,550,878	**7%**	58,913,062	**6%**
bi-256	7,282,095	**22%**	6,321,992	**20%**	57,989,836	**7%**	59,159,275	**6%**

Figure 11: Application-specific branch resolution results

11 bit history register and a 2048 entry second level table and branch target buffer of 2048 entries.

A detailed analysis of all benchmarks has been performed and the set of branches that are highly beneficial for folding have been identified by profiling. A BIT with 16 entries has been assumed; we have targeted 16 branches for the *encode* and 15 for the *decode* of the G.721 benchmarks. For the ADPCM *encoder* we have utilized only 4 branches, and 3 branches for the *decoder*. None of these branches were loop branches, but rather were branches within the tight loop of the corresponding algorithm. Figure 7 shows the statistics for the selected 16 branches for the G.721 encoder. The first row shows how many times the branch is executed; the subsequent rows show the prediction accuracy of the three general purpose predictors described above for each branch. For the G.721 decoder the same set of branches have been selected except for *branch3* (both the decoder and the encoder share the same numerical functions that contain the tight application loops and thus exhibit a very similar behavior). Figures 9 and 10 contain the corresponding information for the ADPCM encoder and decoder, respectively.

Figure 11 shows the results obtained after implementing the proposed application-specific branch resolution logic for early condition evaluation and branch folding. The first column shows the type of auxiliary predictor used for the branches not covered by ASBR. The *not taken* predictor corresponds to essentially having no predictor. Bimodal predictors of sizes 512 (bi-512) and 256 (bi-256) prediction entries have been used with the branch target buffer reduced to a quarter of its size compared to the general-purpose predictors used in the baseline architecture. All simulation results of ASBR show significant performance improvements over the baseline results shown in Figure 6, thus showing that area reductions can be coupled with performance improvements by using the method we propose.

Table 11 is broken down into subtables, corresponding to the encode and decode applications. The first column in each subtable presents the total number of cycles taken to execute the application, while the second column shows the absolute performance improvement. The percentage of the performance improvement for the cases of bi-512 and bi-256 corresponds to an absolute decrease in execution cycles compared to the general-purpose bimodal predictor. The particular predictor compared to is composed of 2048 2-bit saturating counters and 2048 branch target buffer entries, the results for which are presented in the second row of Figure 6.

9. CONCLUSION

In this paper, we present a novel, application-specific customization approach for embedded processor cores. Increasing processor performance and reducing power consumption have been identified as essential goals towards achieving cost-efficient and flexible system implementations. The customization approach we propose herein applied to early branch resolution attacks these goals, utilizing a low-cost, reconfigurable datapath. This customization technique uses a novel approach for transferring application information to the processor micro-architecture and exploiting it dynamically. The ability to re-customize the application-specific branch resolution logic in field is a significant advantage that preserves the flexibility of general-purpose processors. The approach is evaluated on real-life applications and significant performance improvement is shown.

Customizing the processor core with application-specific information promises to be a powerful technique towards higher performance and lower power consumption that allows the processor-centric implementation paradigm and its concomitant advantages to be extended to large classes and parts of complex hardware/software codesign systems implementing various modern applications.

10. REFERENCES

[1] W. H. Wolf, "Hardware-Software Co-Design of Embedded Systems", *Proceedings of the IEEE*, vol. 82, n. 7, pp. 967–989, July 1992.

[2] C. Young and M. D. Smith, "Static correlated branch prediction", *ACM Transactions on Programming Languages and Systems*, vol. 21, pp. 111–159, 1999.

[3] S. McFarling, "Combining branch predictors", Technical Report TN-36, Western Research Laboratory, DEC, June 1993.

[4] S. T. Pan, K. So and J. T. Rahmeh, "Improving the accuracy of dynamic branch prediction using branch correlation", in *ASPLOS V*, pp. 76–84, October 1992.

[5] L. H. Lee, J. Scott, B. Moyer and J. Arends, "Low-cost branch folding for embedded applications with small tight loops", in *32nd MICRO*, pp. 103–111, November 1999.

[6] J. A. Fisher, P. Faraboschi and G. Desoli, "Custom-fit processors: letting applications define architectures", in *29th MICRO*, pp. 324–335, 1996.

[7] J. A. Fisher, "Customized instruction-sets for embedded processors", in *36th DAC*, pp. 253 – 257, June 1999.

[8] C. Lee, M. Potkonjak and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", in *30th MICRO*, pp. 330–335, December 1997.

[9] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0", Technical Report 1342, University of Wisconsin-Madison, CS Department, June 1997.

[10] D. Ditzel and H. McLellan, "Branch folding in the CRISP microprocessor: reducing branch delay to zero", in *14th ISCA*, pp. 2–7, June 1987.

[11] P. B. Gibbons and S. S. Muchnik, "Efficient instruction scheduling for a pipelined processor", in *SIGPLAN*, pp. 11–16, June 1986.

[12] M. S. Lam, "Software pipelining: An effective scheduling technique for VLIW processors", in *SIGPLAN*, pp. 318–328, June 1988.