Automatic Instruction Set Extension and Utilization for Embedded Processors

Armita Peymandoust, Laura Pozzi[†], Paolo Ienne[†], and Giovanni De Micheli

Computer Systems Laboratory
Stanford University
Stanford, CA 94305
{armita, nanni}@stanford.edu

[†]Swiss Federal Institute of Technology Lausanne
Processor Architecture Laboratory
IN-F Ecublens, 1015 Lausanne, Switzerland
{Laura.Pozzi, Paolo.Ienne}@epfl.ch

Abstract

There is a growing demand for application-specific embedded processors in system-on-a-chip designs. Current tools and design methodologies often require designers to manually specialize the processor based on an application. Moreover, the use of the new complex instructions added to the processor is often left to designers' ingenuity. In this paper, we present a solution that automatically groups dataflow operations in the application software as potential new complex instructions. The set of possible instructions is then automatically used for code generation combined with high-level arithmetic optimizations using symbolic algebra. Symbolic arithmetic manipulations provide a novel and effective method for instruction selection that is necessary due to the complexity of the automatically identified instructions. We have used our methodology to automatically add new instructions to Tensilica processors for a set of examples. Our results show that our tools improve designers productivity and efficiently specialize an embedded processor for the given application such that the execution time is greatly improved.

1. Introduction

Embedded multimedia systems are required to be low in cost and high in performance, along with meeting an aggressive time-to-market constraint. Use of application-specific instruction-set processors (ASIP) in such embedded systems is a natural choice as ASIPs have time-to-market advantage over custom design ASICs and performance and power advantages over traditional fixed instruction set processors.

Automating as many steps in the design of application-specific embedded systems is necessary to meet time-to-market requirements. Unfortunately, current available design tools and methodologies cannot meet all designers' needs. Typically, software engineers start with a high level C code that specifies the application and manually specialize the embedded processor such that performance and cost constraints are satisfied. This process starts with profiling the application software to find the computation intensive segments of the code. Mapping these segments to hardware can greatly reduce the execution time of the application. Most base processors are capable of efficiently handling control segments of the application. Thus, the sections that benefit most from acceleration on hardware are data path or basic block segments. Consequently, the application-specific processor is manually tailored to include new ad-hoc functional units and instructions that calculate the computation critical basic blocks of the code. Nevertheless, specialization and design of ad-hoc functional unit extensions can be very lengthy and burdensome, which in turn introduces undesired delay in the overall development process.

In addition, most C compilers are unable to use the new complex instructions of the ASIP efficiently and automatically. In current design methodology, software designers manually insert intrinsic function calls that correspond to the new complex instructions in the computation intensive sections of the code. Manually inserting function calls is both time consuming and error prone. Moreover, designers often miss the opportunity of reusing the new instructions in other sections of the code to further reduce the execution time of the application.

Our objective is to facilitate the specialization of application-specific processors and to automate the use of the new complex instructions added to the processor. In this paper, we propose a new specialization methodology based on extracting multiple-input single-output dataflow graphs and symbolic manipulation of polynomials. First, we automatically identify clusters of combinatorial operations that can be grouped into single operations implemented in new functional units of the ASIP. Next, we use symbolic algebraic algorithms to map dataflow sections of our software to the potential new complex instructions. The combination of algorithms from symbolic computer algebra and standard compiler optimization techniques allows novel automatic code transformations that are hard to find by traditional graph covering methods.

As a motivating example, consider the code segment shown below:

```
int foo(int a, int b, int c, int d){
  return a*b+c*d;
}
```

Assume that for a digital video application `foo` is a critical function. Therefore, we add a functional unit to the application-specific processor that calculates `foo`. Next, consider a basic block of the same digital video application calculating:

```
Y1 = a * R1 + b * G1 + c * B1 + d;
Y2 = a * R2 + b * G2 + c * B2 + d;
Y  = Y2 + q * (Y1 - Y2);
```

With proper variable renaming and the algebraic knowledge that `d = 1*d`, we can calculate `Y1` and `Y2` using the new `foo` instruction added to the processor as follows:

```
Y1 = foo(a, R1, b, G1) + foo(c, B1, d, 1);
Y2 = foo(a, R2, b, G2) + foo(c, B2, d, 1);
```

By using the *expand* routine in symbolic algebra, `Y` can be transformed and mapped to the `foo` instruction:

```
Y = Y2 + q * Y1 – q * Y2;
Y = Y2 + foo(q, Y1, -q, Y2);
```

Thus, the number of instructions used to calculate this basic block is reduced from 15 instructions to 9 instructions that include 5 `foo` instructions, 3 adds, and 1 negate. Now, we formulate one polynomial for the original basic block:

```
Y = a*R2+b*G2+c*B2+d+q*a*R1+q*b*G1+q*c*B1-q*a*R2-q*b*G2-q*c*B2;
```

By applying the *collect* symbolic polynomial manipulation on Y, and substituting `1-q` by `p`, we have:

```
Y = ((1-q)*R2+q*R1)*a+((1-q)*G2+q*G1)*b+((1-q)*B2+q*B1)*c+d;
Y = (p*R2+q*R1)*a+(p*G2+q*G1)*b+(p*B2+q*B1)*c+d;
Y = foo(foo(p,R2,q,R1),a,foo(p,G2,q,G1),b)+foo(foo(p,B2,q,B1),c,d,1);
```

As shown, the new equation can be mapped to 7 instructions: 5 `foo` instructions, 1 add, and 1 subtract. The new mapping is even more efficient than the previous one. This complex solution is hard to find in tree mapping methods since the number of operation to calculate `Y` initially increases. Currently, there is no tool that can perform this kind of algebraic optimizations automatically.

Furthermore, utilization of additional ASIP instructions is the responsibility of the designers. Thus, designers manually implement such optimizations using their knowledge of algebra and insert the proper intrinsic function calls.

This paper presents a methodology that combines detection of potential new instructions for application-specific processors with algebraic manipulations such as the one shown in the previous example. The result of this combination is automatic instruction set selection and mapping. First, a set of potential functional units is extracted from the application software by the multiple-input single-output (MISO) dataflow extraction tool. Next, symbolic algebra is used to map the polynomial representations of the basic blocks to the new instruction set. The new instruction set is determined based on the usage frequency, cost, and possible execution time improvement. Finally, symbolic algorithms are used once more to map the basic blocks to the new instruction set.

The application of this methodology spans from pure ASIP design with extensible functional units to processors equipped with embedded reconfigurable arrays. To minimize microarchitectural and technological assumptions, the analysis of the results focuses on the former type of designs without excluding the viability for the latter implementation. As an example of an extensible ASIP, we are using a Tensilica [1][2] core in our experimental setup.

The paper is organized as follows: Section 2 discusses previous work in software optimization for configurable processors. Section 3 presents our proposed methodology, and explains each of its steps. The results of several examples and the improvements achieved by automatically specializing the Tensilica core are presented in Section 4. Finally, Section 5 summarizes contributions of this work.

2. Related work

This paper combines two related areas of research: automatic identification of instruction-set extensions and use of symbolic algebraic manipulations to map dataflow sections of the code to complex instructions available on the processor. We will discuss the latter area first and then the state-of-the art in instruction identification.

Advanced compilers integrate some tree restructuring capabilities based on algebraic properties to reduce the execution time of complex calculations [3][16]. The goal of such restructuring is typically very precise; for example isolating constants to minimize the amount of address calculation at runtime. In such cases, a number of basic tree transformation rules, applied recursively, result in the desired optimized tree. Similarly, research compilers [4] for SIMD architectures use a fixed set of algebraic tree restructuring rules for associativity and commutativity to improve the quality of SIMD instruction selection. Our approach is more general in that it explores all restructuring possibilities of a dataflow section derived from results of elimination theory and Gröbner bases [15]. A comprehensive set of algebraic manipulations becomes necessary, as defining a straightforward series of transformations is not possible in the general case of complex instruction selection.

The problem of identifying instruction-set extensions consists of detecting clusters of operations that, if implemented as a single complex instruction, maximize a metric—typically performance. Previous works [11][12] have combined template matching (or *instruction mapping*) and template generation (*instruction identification and selection*, in this text) for ASIPs. Kastner et al. [11] cluster operations based on the frequency of node types successions—e.g., multiplications followed by additions. The authors observe that the number of operations per cluster is typically small and conclude that simple pairs of operations appear to be the best candidates. Also, their work does not account for constraints on the number of inputs and outputs of the clusters. Arnold et al. [12] propose a very similar method from the identification perspective, although the overall goal and architectural context is rather different. Work in reconfigurable computing (e.g., [13][14]) also tackles instruction-set identification. Algorithms are relatively simple and typically result in small suboptimal instructions.

3. Our methodology

Here we present a methodology that aims to automate specialization of an application-specific processor by adding new functional units to an extensible basic ASIP core. Our methodology also targets automatic use of the new functional units in the application software. Ideally, designers can profile the software code to find the critical sections of the code. These sections are then added to the ASIP core as new functional units and instructions. In addition, a compiler-like tool would optimize the algorithmic-level description of the software to use the new instructions automatically. However, in reality, optimum selection of new instructions is not possible solely by traditional profiling tools. Moreover, designers need to manually modify their original code to use the new functional units or complex instructions. Usually, designers use the profiling information as a guideline for selecting new instructions. Along with that, the software code is manually restructured to find common blocks or functions that could be mapped to new functional units or instructions. Automating selection and use of new instructions can save much design time.

Figure 1 shows the overview of our two-step methodology. We start with the high-level C code describing our application and the instructions available on the base core. In the first step, we combine multiple-input single-output (MISO) dataflow extraction with symbolic algebraic mapping to define the new instruction set. Note that the base instruction set of extensible ASIPs can generally execute control segments of the application efficiently. Thus, we focus on dataflow sections of the code and add new combinational functional units to the base ASIP to accelerate critical basic blocks of the code.

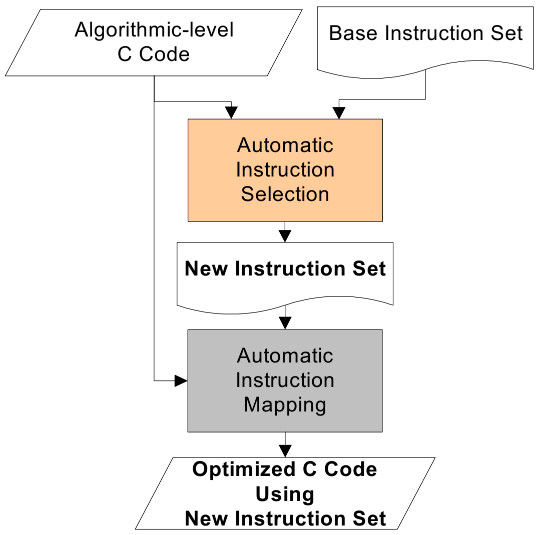
Algorithmic-level C Code
Base Instruction Set
Automatic Instruction Selection
New Instruction Set
Automatic Instruction Mapping
Optimized C Code Using New Instruction Set

Figure 1. Our Methodology

Once the instruction set is selected, decomposition algorithms empowered by symbolic algebra are used in the second step to automatically map the basic blocks to the new instruction set. Each basic block is modeled by its polynomial representation. These polynomials are decomposed into a sequence of instructions available on the new customized ASIP by polynomial manipulation techniques.

Our key contribution is in the use of symbolic algebra combined with dataflow extraction technique to automate instruction set selection and use of the new instruction set for basic block optimization. Note that our methodology is compliant with other software optimization and processor customization techniques. Additional benefits are gained for example by customization of the register file and cache units or by compiler optimizations such as dead code elimination and constant propagation. The next sections describe steps of our methodology in detail.

3.1 Automatic instruction selection

In this section, we will explain the details of the automatic instruction selection step that corresponds to the first shaded box in Figure 1. As mentioned earlier, the base instruction set of an extensible ASIP is usually sufficient for control flow. Therefore, our focus is on adding new instructions that effectively execute critical dataflow sections of the code. Figure 2 shows the different steps necessary for automatic instruction set selection. We start by extracting multiple-input single-output (MISO) dataflow segments from the high-level C code description of our application. The set of extracted MISOs represent the potential new instructions.

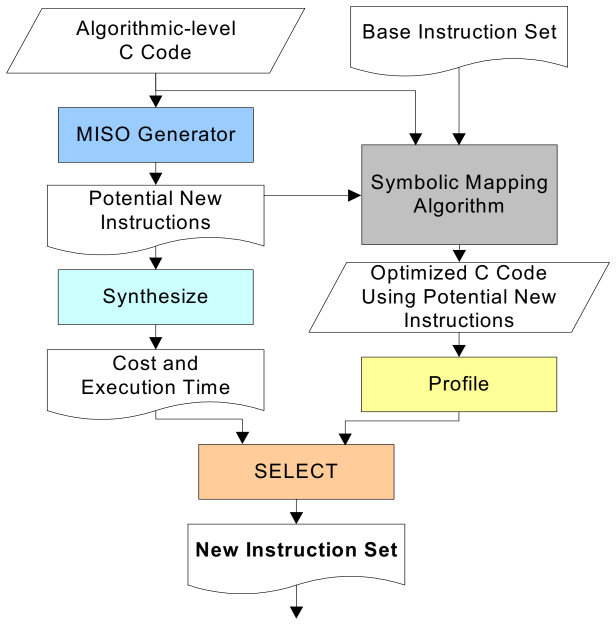

Figure 2. **Automatic**

Instruction Selection

Using symbolic algebraic algorithms, the original software application is optimized and mapped to the union of the base instruction set and the potential new instruction set. The potential new instructions are synthesized in order to extract their cost and execution time. The next step is to profile the optimized software code. The output of the profiler indicates the frequency that each potential new instruction can be used in the application. Note that the symbolic decomposition step automatically identifies all possible sections of the code that can be mapped to extracted MISOs without manual intervention or the need to restructure the original program. Using the frequency of each instruction and their associated cost and execution time, a set of most promising MISOs are selected as instructions to be added to the base ASIP. In the next section, we will describe the MISO extraction and symbolic decomposition steps in more detail.

3.1.1 MISO extraction

The multiple-input single-output (MISO) dataflow extractor tool described in this paper implements two different algorithms. Both algorithms extract single output subgraphs from the basic blocks of the embedded application that correspond to potential new instructions. The analysis starts with the directed acyclic graph (DAG) representation of the basic blocks of the given application. Nodes of the DAG are assembler-like instructions and edges represent data dependency among instructions.

The first method is a greedy algorithm of linear complexity that extracts maximal single output subgraphs from basic blocks. The extracted subgraphs are called *MaxMISO*s (maximal multiple-input single-output). The algorithm starts from each exit node of the basic block and constructs a subgraph by trying to recursively include the parent nodes [9]. Subgraph formation never stops for excess of inputs—inputs are unlimited in a MaxMISO—but it stops if inclusion of a further parent node violates the output constraint. Therefore, MaxMISOs are maximal in the sense that adding any further node would fundamentally violate the single output constraint. The algorithm complexity is linear in the number of nodes of the initial graph. Extracted subgraphs produce a single result, while their number of inputs is unlimited. If the resulting number of inputs is unpractical, the MaxMISO is ignored. The MaxMISO algorithm represents a good tradeoff between complexity of exploration and effectiveness of the resulting extracted instructions.

The second algorithm used in this paper is called *Optimal* [10]. It extracts instructions satisfying user-given input/output constraints that result in maximal speedup. *Optimal* analyses the dataflow graphs of the basic blocks of the application and considers all possible subgraphs. The input and output requirement of the subgraphs is calculated and only those satisfying all constraints are selected for further consideration. The number of subgraphs being exponential in the number of nodes of the graph, *Optimal* has an exponential worst case complexity; yet, it exploits some graph characteristics which allow significant pruning of the search space and, in practice, it exhibits a subexponential complexity. Graphs with up to a couple of hundreds of nodes can be processed in matter of hours. While *Optimal* is designed to satisfy any user-given output constraint, it has been used here only with a single output.

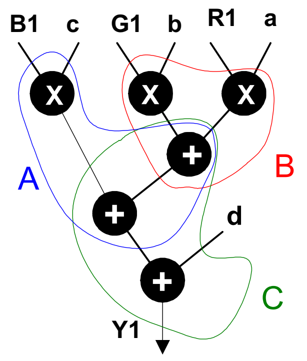

Figure 3. MISO Extraction on Y1 = a*R1 + b*G1 + c*B1 + d

Speedup estimation is then performed for the potential instructions extracted by either the *MaxMISO* or the *Optimal* algorithm. The estimation consists in comparing the approximate subgraph execution time in software, as a sequence of instructions, with the approximate time the subgraph takes if implemented in hardware, as a single special instruction. The most promising candidates are then passed on to the *Symbolic Mapping* phase.

The behavior of *Optimal* is now shown on the motivational example seen in Section 1, for an input/output constraint of 4/1. Part of the DAG of the main basic block is shown in Figure 3. *Optimal* identifies subgraphs within constraints, and estimates their gain by using a rough estimation model, described in [10]. The subgraphs A, B and C, are finally selected by *Optimal* to be passed on to the next phase, as the most promising candidates for new instructions. Candidate B corresponds to the `foo` instruction shown in Section 1. Next, we will show how Candidate B is chosen by the symbolic mapping technique as a new instruction.

3.1.2 Symbolic mapping and optimization

The symbolic mapping algorithm requires two sets of inputs: a set of polynomials representing the basic blocks of the application and another set of polynomials representing the complex dataflow instructions. The goal of the symbolic optimization step is to decompose the polynomial representations of the basic blocks into a minimum number of polynomial representations of available instructions. Such decomposition is done with the help of symbolic computer algebra routines and algorithms. Functional units added to an extensible ASIP execute in one cycle or they are automatically pipelined [2]. Thus, using a minimum number of instructions to calculate a given basic block improves its execution time.

The polynomial representation of a basic block can be directly extracted from the C code if the basic block calculates a polynomial function. If the basic block performs a series of bit manipulations or Boolean functions, interpolation-based algorithms [8] can be used to formulate the equivalent polynomial representation. Note that Boolean functions and polynomial functions accelerate greatly when mapped to hardware. Therefore, these basic blocks are the excellent candidates to be mapped in new functional units of the processor. Approximation, such as Taylor or Chebyshev series expansion, can also be used to extract polynomial representation for basic blocks that calculate a transcendental function. In this paper, we will not use approximation techniques.

Symbolic computer algebra is a set of algorithms capable of algebraic manipulation of expressions containing undetermined values (symbols), such as variable `x` in `(x+1)*(x-1)`. Several commercial symbolic computer algebra softwares are available on the market; Maple [5] and Mathematica [6] are most widely used. Most interesting symbolic polynomial manipulations are based on Gröbner bases [7]. Gröbner bases also solve variable elimination in a set of polynomials and ideal membership problems, which is the core of the simplification modulo set of polynomials [7]. This section gives a brief overview of the mapping algorithm. The core of the library-mapping algorithm is the simplification modulo set of polynomials (*simplify*) routine. The polynomial representations of the basic blocks are simplified modulo a set of polynomials. This set, a.k.a. the *side relation set*, represents a subset of the instruction set. Next, we describe the *simplify* routine by means of the example in Section 1. The polynomial representation of the basic block is equal to (`p=q-1`):

```
Y=a*q*R1+a*p*R2+b*q*G1+b*p*G2+c*q*B1+c*p*B2;
```

Side relation sets are selected from the polynomial representations of the MISOs reported by the MISO extraction tool (Section 3.1.1) with proper variable renaming. In this example, we show two side relation sets, `siderel` and `siderel2`, with elements whose polynomial representations match Candidate B shown in Figure 3. Next, we apply simplify on `Y` modulo the selected side relation sets. The result reported by Maple is shown in bold-italic. A mapping is found when the result of simplify is only one variable or corresponds to a polynomial representation of an instruction in our instruction set. This process is repeated for different combinations of the MISOs and the instructions available on the base core. Candidate B is chosen as a new instruction as it is more frequently used in execution (or mapping) of `Y`.

```
> siderel:={s1=q*R1+p*R2, s2=q*G1+p*G2, s3=q*B1+p*B2};
> z := simplify(Y, siderel, [R1, R2, G1, G2, B1, B2]);
    z = a*s1+b*s2+c*s3
> siderel2:={s4=s1*a+s2*b, s5=s4*1+c*s3};
> simplify(z, siderel2, [s1, s2, s3]);
    s5
```

As shown, side relation set selection is a non-trivial task. Therefore, to find the best possible mapping, the side relation set should be set equal to all subsets of the instruction set with all possible permutations of the input variables. Algorithm 1 is used to prune the search space efficiently. Let `S` be the polynomial representation of the basic block to be decomposed into complex dataflow instructions. We start by simplifying `S` modulo each instruction as the side relation. The simplification results are stored in a tree data structure. If a simplification result is identical to the polynomial representation of an available instruction, a possible solution is found and the corresponding tree node is marked accordingly. If the simplification result stored in a tree node does not correspond to a library element, we recursively apply the same steps to the new tree node.

Algorithm 1. Decompose *S* into the instruction set *L*

```
procedure Decompose($S$, $L$)
  # Given a polynomial representation of the basic block $S$
  # and a set of polynomials $L$ corresponding the instruction set
  # decompose $S$ into elements of library $L$.

  # initialize tree
  treeroot($S$);
  $depth \leftarrow 0$
  $bound \leftarrow -1$
  while depth $\neq$ bound do {
    $bound \leftarrow$ Explore($S$, $L$, $depth$) # Explore is defined below
    $depth \leftarrow depth + 1$
  }
  report best solution in tree
end

# used in Decompose procedure
int function Explore($S$, $L$, $d$)
  $bound \leftarrow -1$
  for all $n \in$ in tree with depth $d$ do{
    for all $sr \in L$ do{
      $result$ = simplify($n$, $sr$);
        # make $result$ a child of node $n$
      addchild($n$, $result$);
        if $result \in L$
          # solution is found
          $bound$ = treedepth($result$);  }}
  # returns –1 if no solution is found yet.
  return($bound$)
end
```

The bounding function used to reduce the search space is the number of instructions used to calculate the basic block. In other words, if we find a solution that calculates the basic block with two instructions we will not explore solutions requiring more than two instructions. Nevertheless, we will uncover all two-instruction solutions and choose the one with optimal cost or execution time. The number of instructions used is equivalent to the depth of the simplification tree. Therefore, the tree is bounded by the depth of the first solution found. This algorithm was implemented in C with calls to Maple V for symbolic manipulations.

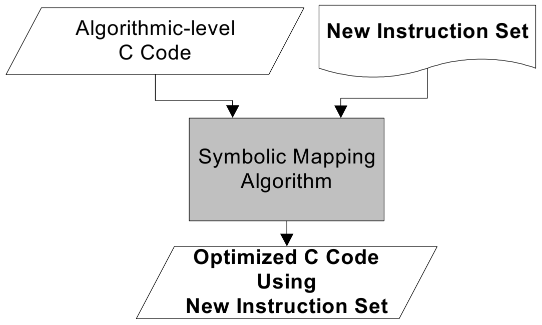
Algorithmic-level C Code
New Instruction Set
Symbolic Mapping Algorithm
Optimized C Code Using New Instruction Set

Figure 4. Automatic Instruction Mapping

3.2 Automatic instruction mapping

The new instruction set of the ASIP has been chosen by the step described earlier. The original software code is now automatically transformed to use the new instruction set assisted by symbolic polynomial manipulation algorithms. Figure 4 gives an overview to the automatic instruction-mapping step. This step also corresponds to the second shaded box of Figure 1. The polynomial representations of basic blocks of the software application and the new instruction set of the ASIP are available to the symbolic mapping algorithm. As opposed to tree covering based algorithms, in our algorithm, mapping is performed simultaneously with algebraic manipulations.

The automatic instruction-mapping step uses Algorithm 1 described in Section 3.1.2. The output of this step is optimized C code with intrinsic function calls automatically inserted. The optimization criteria consist of using a minimum number of instructions to calculate a basic block of the original code. Since added functional units either are pipelined or execute in one cycle, this mapping greatly reduces the execution time.

4. Results

We have optimized several Tensilica [2] cores for a set of software examples using our automatic instruction selection and mapping methodology. In the first step, the MISO extraction tool selects a set of possible complex instructions for each software application. The symbolic mapping technique is used to map the code to the new instructions available. At the end of this step, a subset of the MISO set is selected and implemented as new functional units of the ASIP core under design. The selection is based on the cost of each MISO and the frequency of its use.

In the next step of the flow, we take the new instruction set and automatically use the complex instructions available to optimize our original software code. We have used the Tensilica [2] software toolchain to measure the execution time improvement of each of our examples as result of the new instructions added to the base core. The base core is the default 32-bit Tensilica core plus a 32-bit multiplier. The functional units selected are added to the base core using the Tensilica instruction extension (TIE) language. Table 1 reports the execution time of each example (measured in cycles) as reported by the Tensilica instruction set simulator.

The first five examples in Table 1 are simple filters and dot-product examples from a DSP benchmark. The *DES crypt* example is the MD5 message-digest algorithm that produces a 128-bit fingerprint for an arbitrary length message or file. The *adpcm* example is an adaptive differential pulse code demodulator software used for speech compression/decompression. Finally, the last example is a fixed-point MP3 decoder software decoding a 5 second long stream. By applying our methodology to the MP3 decoder and adding only three new instructions to the base core, the decoder executes three times faster.

Examples	**Base core Execution time (cycles)**	**Extended core Execution time (cycles)**	**Improvement (%)**
dot_product	72990	54011	26.00
iir	88838	20652	76.75
fir_2dim	168978	114488	32.25
convolution	182492	123035	32.58
fir	268228	158642	40.86
DES crypt	1118570	916884	18.03
adpcm	22514517	14176587	37.03
MP3	2224094335	745248522	66.49
Average			41.25

Table 1. Execution time improvements reported by the ISS

To estimate the cost associated with the execution improvements reported in Table 1, we have synthesized the new functional units added for each example using Synopsys Design Compiler and a 0.35-micron CMOS technology library. The area of the base core is approximately 0.29 mm^2 in this technology. Table 2 shows the number of new instructions added to the base core, the area of the new instructions, and the area increase of the base core. As it is observed from Table 2, our methodology selects a small number of instructions to be added to the base processor that result in modest area increase. Nevertheless, due to its strong instruction selection and mapping engine, the instructions added are key instructions that can be used in many sections of the code, and thus significantly decrease the execution time. Note that the area reported in Table 2 is an upper bound, as the new instructions are synthesized separate from the base core and resource sharing is not considered. For all examples shown in this section, we have added a total of ten different instructions to different cores. The complexity of the added instructions ranges from two operations to twenty operations.

Examples	**Number of Instructions Added**	**Area of Instructions Added (mm^2)**	**Area Increase of the Base Core (%)**
dot_product	1	0.211	7.3
iir	1	0.503	17.3
fir_2dim	1	0.211	7.3
convolution	1	0.211	7.3
fir	1	0.199	6.8

DES crypt	3	0.142	4.9
adpcm	1	0.103	3.5
MP3	3	0.564	19.5
Average		0.268	9.2

Table 2. Area cost of the added instructions

5. Conclusion

The contribution of this paper is a new methodology that automates the selection of very complex instruction set extensions for ASIPs together with aggressive techniques to map the basic blocks to such complex instructions. This work focuses on arithmetic intensive applications such as multi-media processing. A basic ASIP core is extended automatically to include ad-hoc functional units that accelerate the dataflow sections of the software application. A set of potential instructions is generated by the multiple-output single-input (MISO) dataflow extraction tool. Symbolic computer algebra is used to discover transformations that expose unintuitive opportunities for mapping basic blocks of an application into the potential instructions. The most frequently used MISOs by the symbolic mapping tool are selected and added to the base ASIP processor. Symbolic algebra automates very smart instruction mapping previously only possible by designer's manual intervention.

We demonstrate the application of our tool to a set of arithmetic intensive examples including an MP3 decoder software. A Tensilica core was optimized for each application using the Tensilica tool set. We have achieved an average of 41% improvement in the execution time of our examples, while paying only an average of 9.2% penalty in area cost.

Another possible application of our technique is to facilitate reuse of an ASIP in future generations of an application. While hard-wired ad-hoc functional units present the risk of inflexibilities towards subsequent changes of an application, our smart symbolic mapping techniques increase the possibility of using instructions tailored for a previous generation of the application. In future work, we also plan to find dataflow instructions with more than one output. Such sections can be selected by the *Optimal* [10] algorithm and represented by a set of polynomials for the symbolic mapping step.

6. References

[1] A. Wang, E. Killian, D. Maydan, and C. Rowen, "Hardware/Software Instruction Set Configurability for System-on-Chip Processors", *Proceedings of the 38th Design Automation Conference*, June 2001.

[2] Tensilica Inc., http://www.tensilica.com, 1997.

[3] S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufmann, 1997.

[4] R. Leupers, Code Optimization Techniques for Embedded Processors, Kluwer Academic, 2000.

[5] Maple, Waterloo Maple Inc., http://www.maplesoft.com, 1988.

[6] Mathematica, Wolfram Research Inc., http://www.wri.com, 1987.

[7] T. Becker and V. Weispfenning, Gröbner Bases, Springer-Verlag, New York, NY, 1993.

[8] J. Smith and G. De Micheli, "Polynomial circuit models for component matching in high-level synthesis", *IEEE Trans. on VLSI*, Vol. 9, Issue 6, Dec. 2001, pp 783 –800.

[9] C. Alippi, W. Fornaciari, L. Pozzi, and M. G. Sami, "A DAG based design approach for reconfigurable VLIW processors", *Proceedings of the DATE Conference*, pp. 778-780, 1999.

[10] K. Atasu, L. Pozzi, and P. Ienne, "Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints", *Proceedings of 40th Design Automation Conference*, Anaheim, CA, June 2003.

[11] R. Kastner, A. Kaplan, S. Memik, and E. Bozorgzadeh, "Instruction generation for hybrid reconfigurable systems", *ACM Transactions on Design Automation of Embedded Systems (TODAES)*, October, 2002.

[12] M. Arnold and H. Corporaal, "Designing domain specific processors", *Proceedings of the 9th International Workshop on Hardware/Software CoDesign*, pages 61–66, Copenhagen, April 2001.

[13] B. Kastrup, A. Bink, and J. Hoogerbrugge, "ConCISe: A compiler-driven CPLD-based instruction set accelerator", *Proceedings of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 1999.

[14] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit", *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 225–235, June 2000.

[15] D. Cox, J. Little, and D. O'shea, "Ideals, Varieties, and algorithms", Springer-Verlag, New York, NY, 1997.

[16] J. Zory and F. Coelho, "Using Algebraic Transformations to Optimize Expression Evaluation in Scientific Code", *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques,* October 1998.