ABSTRACT

The ST120 Digital Signal Processor - Micro-Controller Unit (DSP–MCU) core was designed by STMicroelectronics in order to meet the ever-increasing digital signal processing requirements of portable and consumer applications. Like other recent high-end DSP–MCU cores, the ST120 blends traditional DSP features with modern Instruction-Level Parallelism (ILP) capabilities.

Compiler management of the ST120 features presents a unique challenge to the code generation. The ST120 Linear Assembly Optimizer (LAO) effectively exploits instruction-level parallelism, while enabling compact code size. In this paper, we focus on the LAO implementation of the SSA representation, the IF-conversion, the SLIW scheduling, and the LAO improvements to register allocation. This includes solutions to problems that arise when compiler optimizations are applied to assembly-level, already predicated code.

1. INTRODUCTION

Processors for portable applications are required to deliver high-performance digital signal processing at low power. In addition, the ROMs that contain application codes significantly contribute to system power consumption, so processors are also required to expose a compact instruction set.

Traditional means of achieving high processor performance, such as superscalar implementation with branch prediction, out-of-order execution, and large register sets that consume instruction encoding space, cannot be used when both energy efficiency and code size are of primary concern. On the software side, use of Instruction-Level Parallelism (ILP) enhancing techniques such as loop unrolling, software pipelining [33] with modulo expansion [26], or branch straightening with tail duplication, is severely constrained by the near-zero tolerance of portable applications to code size increase.

The ST120 core addresses these problems by decoupling the address computations from the data computations. Decoupled implementation consumes less energy and takes less die area than an out-of-order superscalar implementation, yet retains some of its important benefits. In particular, access-execute decoupling makes loads to data registers execute with an apparent zero RAW latency, thus obviating the need for software pipelining and modulo renaming on most digital signal processing kernels. The ST120 core also supports an advanced predication model that includes branch shadow execution. Inside a branch shadow, predicated instructions may execute before the branch is resolved. This is especially effective for conditional branches that cannot be removed by predication, such as loop early exits.

In this paper, we first introduce the ST120 architecture and micro-architecture features that challenge code generators. We then describe our approach to SSA optimization, predication, instruction scheduling, and register allocation, as implemented in our ST120 Linear Assembly Optimizer (LAO) tool. In the last section, we demonstrate the efficiency of our techniques by measuring how performance and code size are affected by running the LAO on the output of the best available ST120 production C compilers.

2. THE ST120 DSP-MCU CORE

2.1 ST120 Architecture Overview

The STMicroelectronics ST120 [34] is a Digital Signal Processor – Micro Controller Unit (DSP–MCU) core, designed for digital signal processing in telecommunication applications. In order to effectively support C/C++ compilers, DSP-MCUs combine traditional DSP features with VLIW and EPIC principles [16, 35, 12]. In particular, the ST120 Instruction Set Architecture (ISA) is a predicated load-store architecture, that defines three register sets (figure 1):

Figure 1: The ST120 architectural registers.
Data Registers This set contains 16 40-bit registers, any of which can be used as accumulators by the multiply-accumulate instructions.

Address Registers This set contains 16 32-bit registers, whose main purpose is to provide the base and offset of memory accesses.

Control Registers This set includes all the specialized registers, including the Guard Register (GR) that contains the guards (predicate registers) used for instruction predication, and three hardware loop register sets.

Architectural DSP support includes multiply-accumulate instructions that multiply 16-bit sub-registers and accumulate into 40 bits using signed, unsigned, or fractional arithmetic [12]. Other data instructions uniformly support 16-bit packed, 32-bit, and 40-bit, arithmetic for signed, unsigned, or fractional data types. Unlike older DSPs, the ST120 architecture does not introduce mode bits, as arithmetic types and rounding modes are explicit in the instructions.

The ST120 addressing modes include auto-modification of base address pointers. Post-modification is the default case, although one addressing mode allows pre-modification for implementing software FIFOs. Other DSP addressing modes include modulo addressing, and bit-reversed addressing. Hardware loops enable the ST120 to iterate a block of instructions as a counted loop without explicit decrementing, testing, and branching. The ST120 also supports uncounted (infinite) hardware loops.

The ST120 ISA defines two main instruction modes, respectively called GP16, and GP32. In the GP32 mode, instructions are encoded into 32-bit, can have up to four register operands, and may only access limited architectural resources. In the GP16 mode, instructions are encoded into 16 bits, have two register operands (except ADD / SUB), and may only access limited architectural resources. The purpose of the GP16 mode is to enable compact code size on microcontroller tasks, following the principles pioneered by the ARM architecture with its Thumb instruction mode [1].

The two ST120 instruction modes share the same Application Binary Interface (ABI), and mode switching automatically takes place at function boundaries, depending on the alignment of the called function. A function byte address aligned 0 mod 4 triggers the GP32 instruction mode, while a function address aligned 2 mod 4 triggers the GP16 instruction mode. Within a function, explicit mode switching is also possible through dedicated instructions.

2.2 The ST120 Decoupled Implementation

The ST120 core implements access-execute decoupling [36], following the same design principles as in the floating-point unit of the MIPS R8000 [22]. In the GP16 and GP32 instruction modes, execution is in-order two-way superscalar. Aligned pairs of instructions are simultaneously decoded and expanded by the Control Unit (CU) into micro-instructions, which are sent to the dispatch queues of four parallel execution pipelines: two for the Address Unit / General Unit (AU/GU), and two for the Data Unit (DU) (figure 2).

The ST120 decoupled implementation is best illustrated on loads to DU registers. For each DU load, the CU generates two micro-instructions: the effective address computation, which is dispatched to one of the AU/GU execution pipelines; the receive of the loaded value, which is dispatched to one of the DU execution pipelines. (Selection of the execution pipelines inside the AU/GU and the DU only depends on the static alignment of the load instruction.) The receive of the loaded value, and the execution of the subsequent DU micro-instructions, wait for the data to return from memory. The DU dispatch queues are deep enough to prevent these delays of DU execution from stalling the CU.

From the programmer’s point of view, a DU load appears to return data in zero cycles to RAW dependent instructions. This feature is especially useful on a DSP–MCU, as it decreases the need for architectural registers, and also reduces the code size. In particular, software pipelining is not required for most digital signal processing kernels. On the other hand, the DU loads are blocking, meaning that the DU instructions that follow a DU load execute after the loaded value returns from memory, even if they do not use this value. Loads to the AU or the CU registers are non-blocking, but they do expose the full memory latency.

In addition to the GP16 and GP32 instruction modes, the ST120 core implements a Scoreboarded Long Instruction Word (SLIW) instruction mode, where “instruction bundles” that are comprised of two AU/GU GP32 instructions, and two DU GP32 instructions, dispatch every cycle.

In the SLIW mode, the data dependences are scoreboarded, provided they hold between instructions that belong to different bundles. Scoreboarding means that dynamic delays are inserted whenever required to enforce the dependences. Inside a bundle, the data dependences are not scoreboarded, but the zero-latency dependences are enforced, in particular the RAW dependences that originate from the DU loads. Both the inter-bundle scoreboarding and the intra-bundle zero-latency dependences significantly reduce the size of the SLIW code, compared to the equivalent VLIW code.

In the memory layout of a SLIWB bundle, the two AU/GU instructions always come first, followed by the two DU instructions. However, in order to make the SLIW mode more friendly in assembly language, the four GP32 instructions that make a SLIWB bundle do not always appear in this memory order. Precisely, the ST120 assembler syntax defines the following SLIWB “groupings” (SLIWB bundle templates):

<table>
<thead>
<tr>
<th>Group</th>
<th>Instr. 1</th>
<th>Instr. 2</th>
<th>Instr. 3</th>
<th>Instr. 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Load</td>
<td>Load</td>
<td>DU-op</td>
<td>DU-op</td>
</tr>
<tr>
<td>1</td>
<td>Load</td>
<td>DU-op</td>
<td>DU-op</td>
<td>Branch</td>
</tr>
<tr>
<td>2</td>
<td>Load</td>
<td>DU-op</td>
<td>DU-op</td>
<td>Store[AU-op]</td>
</tr>
<tr>
<td>3</td>
<td>DU-op</td>
<td>DU-op</td>
<td>Store[AU-op]</td>
<td>Store[AU-op]</td>
</tr>
<tr>
<td>4</td>
<td>DU-op</td>
<td>DU-op</td>
<td>Store[AU-op]</td>
<td>Branch</td>
</tr>
</tbody>
</table>

Figure 2: The ST120 decoupled implementation.
The purpose of these SLIW groupings is to enforce an order of the instructions within a SLIW bundle where the intra-bundle zero-latency RAW dependences are ordered from left to right. For the compiler however, assembly reordering of the instructions is merely cosmetic. The real challenge is to exploit the zero-latency intra-bundle dependences, which include RAW, WAR, and WAW, data dependences.

2.3 Predication and Branch Shadow

Modern architectural support for predicated execution is defined by the IMPACT-EPIC architecture, itself generalized from the Cydra-5 and PlayDoh architectures [2]. This architectural support, later referred to as the “IMPACT” predication model, introduces Predicate Define Instructions (PDIs) with the syntax: \( \text{G}_y \cdot \text{G}_z \). Here, \( \text{G}_x \) is the source predicate register, \( \text{G}_y \) and \( \text{G}_z \) are the destination predicate registers, and \( \text{type}_0 \) and \( \text{type}_1 \) are the predicate types defined as follows:

- **Unconditional** Destination is written \( \text{G}_x \land (\text{R}_n \ \text{cond} \ \text{R}_p) \).
- **Parallel-AND** Destination is written \( \text{G}_x \land (\text{R}_n \ \text{cond} \ \text{R}_p) \).
- **Parallel-OR** Destination is written \( \text{G}_x \lor (\text{R}_n \ \text{cond} \ \text{R}_p) \).
- **Normal** Destination is written \( (\text{R}_n \ \text{cond} \ \text{R}_p) \).
- **Disjunctive** Destination is written \( (\text{R}_n \ \text{cond} \ \text{R}_p) \).
- **Conjunctive** Destination is written \( (\text{R}_n \ \text{cond} \ \text{R}_p) \).

The ST120 architecture allows almost all the GP32 instructions (including those in SLIW bundles) to be predicated. Predicate values are taken from the Guard Register, which contains 16 individual guards (guard \( \text{G}_{15} \) is always \( \text{true} \)). These guards are set by the Guard Modification Instructions (GMIs). Compared to the IMPACT model [2], the distinguishing features of the ST120 predication are:

- A ST120 GMI may only define one guard. This scheme is easier to manage by the compiler predication algorithms, while losing little expressive power under techniques like the Program Decision Logic predication [2].
- All the ST120 GMIs are guardable, that is, they correspond to the Normal IMPACT PDIs. Unconditional PDIs that are the default on IMPACT are not so useful on the ST120 because non-guardable effects are not compatible with branch shadow execution (see below).
- The ST120 GMIs include “Guard Combination Instructions” to perform logical operations (AND, OR, XOR, etc.) between individual guards. This fills the need for the Conjunctive and Disjunctive PDIs, that have been added recently to the IMPACT model [2].
- The ST120 GMIs do not include the Parallel-AND and Parallel-OR instructions, due to instruction encoding space constraints, and given their limited performance benefits for the issue width of two DU instructions.
- The ST120 conditional branches are taken when their guard is \( \text{false} \). Non-branch instructions are effectively executed when their guard is \( \text{true} \). The main reason for this is the support of branch shadow execution.

Branch shadow execution is an innovative ST120 feature whose purpose is to eliminate the not-taken conditional branch penalty. Use of the ST120 branch shadow is as follows:

![Figure 3: The LAO internal phases.](image_url)

### 3. Code Generator Optimizations

#### 3.1 The ST120 Linear Assembly Optimizer

A number of ST120 features challenge traditional compiler code generators. These features include the split AU – DU register file, the decoupled implementation, the SLIW instruction mode, the predication model with branch shadow, the DSP addressing modes, and the DSP arithmetic support. Compiler exploitation of these features is demonstrated by the ST120 Linear Assembly Optimizer (LAO).

The purpose of the LAO is to convert a program written in Linear Assembly Input (LAI) language to the ST120 basic assembly language that is suitable for assembly, linking, and execution. Although initially designed for the ST120 assembly language programmers, the LAO has proved itself very effective at improving the ST120 C compiler output.

The LAI language is a superset of the GP32 assembly language, extended by the ST120 assembly level intrinsic functions, and where symbolic register names can be freely used. From this input, the LAO produces a program where the GP32 and SLIW instruction-level parallelism is exploited, the ST120 intrinsic functions are expanded, and the symbolic registers are mapped to architectural registers.

The different LAO processing phases are depicted in figure 3. After initial macro-processing and parsing of a LAI source file, an Assembler Intermediate Representation (AIR), which basically amounts to code and data streams, is built. This AIR is then lifted to a LAO Intermediate Representation (LIR), where the function boundaries and the data segments are identified, and where all the instruction effects are exposed. From there, some of the most important code generator optimizations that are applied are:
Loop Analysis Based on the DJ-graph technique [37].

Range Optimizations Collect integer ranges, and remove redundant sign extensions.

Global Optimizations Include constant propagation, dead code elimination, and expression simplification.

Loop Restructuring Unrolls loops, and maps inner loops to counted or uncounted hardware loops.

Address Optimizations Exploit the DSP addressing modes, and balance expressions between the DU and the AU.

Pattern Optimizations Recognize the DSP instruction patterns, and substitute more efficient code.

Intrinsic Expansion Context-sensitive macro expansion, including the optimization of division by constants.

IF-conversion Predication of single-entry control-flow regions into superblocks (§3.3).

Prepass Scheduling Superblock scheduling, and software pipelining with modulo scheduling (§3.4).

Register Allocation Effective register assignment and spilling of predicated code (§3.5).

Postpass Scheduling Superblock scheduling, branch shadow exploitation, and SLIW code linearization (§3.4).

Following these optimizations, the LIR is mapped back to the AIR, which is then emitted as assembly code.

In the remainder of this section, we focus on the LAO implementation of the Single Static Assignment (SSA) representation of the program, the IF-conversion phase, the instruction scheduling phase, and the register allocation phase.

The main purpose of the SSA representation in the LAO is to enable efficient global and loop analyses, and to ease implementation of the related optimizations. SSA-based analyses include loop induction variables, integer range propagations, and simple alias analysis. SSA-based optimizations include constant propagation, dead code elimination, useless sign extension removal, expression simplification, generation of auto-modification addressing modes, and expression balancing between the DU and the AU. Another benefit of the SSA representation is that it renaming many multiple register definitions, so the instruction scheduler is significantly less constrained by WAR and WAW register dependences.

Advanced IF-conversion is required on the ST120 core to optimize both code size and code performance. As the ST120 does not implement branch prediction, GOTO instructions execute in several cycles, even when the branch is unconditional, or conditional not-taken. By using the hardware loops, all performance-critical looping branches are converted to early exit branches that are mostly not taken. The cost of early exit branches is minimized thanks to branch shadow exploitation. On other conditional branches, IF-conversion also yields large improvements as it eliminates GOTO instructions, and because it enables the creation of larger scheduling regions.

Instruction scheduling is critical for code performance on the ST120 core, due to its issue width of 4 instructions in the SLIW mode, and because of the long register flow latencies that appear when moving values from the DU to the AU/GU. The decoupled implementation of the ST120 core creates other peculiarities. First, loads to the DU registers are blocking, a feature that challenges traditional instruction schedulers. Second, the zero-latencies must be exploited in order to benefit from the decoupled implementation. Third, the instruction scheduler has to exploit the branch shadow.

Efficient register allocation is crucial on the ST120, that has a relatively small number of registers arranged in separate register files. The LAO register allocator is designed to efficiently allocate registers and generate spill code for predicated code, while reducing the number of MOVE operations through the application of the repeated register coalescing. The repeated register coalescing is also used during the conversion from the SSA representation, see §3.2.

3.2 The LAO SSA Representation

The Single Static Assignment (SSA) representation of programs enables efficient global analysis and optimizations [17, 39, 5]. In the LAO, a problem arises as the SSA representation has to deal with already predicated operations that may come from the LAI code. This requires to support predicated code in the LAO SSA representation, even though the LAO IF-conversion is performed after the SSA optimizations. The problem is that the SSA representation does not express the predicated definitions merge points.

To address this issue, on each predicated operation, we add an implicit use of the previous (perhaps predicated) definition of the defined register. We attach the following meaning to an implicit use: the defined register takes the result of the evaluation of the operation if this operation’s predicate is true, otherwise it takes the result of the previous reaching definition. This allows to rename definitions that affect the same register under different predicates, and to perform safe SSA optimizations on predicated code.

When translating the program back from the SSA representation, we have to express the fact that the two registers (the predicated definition and its implicit use) must be renamed to the same register. We implemented the SSA translation algorithm from [27], that provides a way to handle two-operand machine instructions, with the use and the definition sharing the same register. Another advantage of the algorithm from [27] is that it enables the LAO implementation to correctly enforce the register targeting constraints of the ST120 Application Binary Interface (ABI).

On the other hand, we found that such use of the algorithm from [27] generates too many MOVE operations: not only MOVEs are generated from translating the Φ-operations, but also one MOVE is generated for each predicated definition. These MOVEs are eliminated on the regions selected for IF-conversion and software pipelining by applying the repeated register coalescing algorithm (see §3.5) before prepass scheduling. The MOVE elimination is important at this point since otherwise both the IF-conversion and the prepass instruction scheduler would have to deal with them.

3.3 The LAO IF-Conversion

The LAO implementation of the IF-conversion is based on Fang’s algorithm [15]. This algorithm allows the IF-conversion of single-entry acyclic control flow-regions, based on three steps: the first step assigns a control predicate to each basic block in the region to be predicated; the second step assigns predicates to the region’s individual operations, with on-the-fly predicate promotion; in the third step, intra-
Foreach operation ∈ block
    
    \[ \text{G}_p \leftarrow \text{getAssignedPredicate(operation)} \]
    
    if \text{isNotPredicated(operation)}
    
    \[ \text{operation.predicate} \leftarrow \text{G}_p \]
    
    else
    
    \[ \text{G}_s \leftarrow \text{operation.predicate} \]
    
    if existsLocalRenaming(\text{G}_s, \text{G}_s)
    
    \[ \text{G}_s \leftarrow \text{getLocalRenaming}(\text{G}_s, \text{G}_p) \]
    
    else
    
    \[ \text{G}_s \leftarrow \text{newRegister()} \]
    
    if existsLocalDefinition(\text{G}_s)
    
    \[ \text{localOptimize(operation, } \text{G}_s, \text{G}_s', \text{G}_p) \]
    
    else
    
    if isConditionalExitBranch(operation)
    
    \[ \text{insertOperation}(\text{G}_s' \leftarrow \text{G}_s \lor \neg \text{G}_p) \]
    
    else
    
    \[ \text{insertOperation}(\text{G}_s' \leftarrow \text{G}_s \land \text{G}_p) \]
    
    end if
    
    end if
    
    \[ \text{setLocalRenaming}(\text{G}_s, \text{G}_p, \text{G}_s') \]
    
    end if
    
    if isPredicateDefinition(operation)
    
    \[ \text{killLocalRenaming}(\text{G}_s) \]
    
    end if
    
end foreach

Figure 4: Local predicate assignment in the LAO.

region branches are removed, and the machine code that computes the control predicates is created.

Fang’s algorithm has a number of interesting properties: it minimizes the number of control predicates; it inserts predicate definitions as high as possible in the dominator tree; and it control-speculates operations by predicate promotion. The IMPACT-1 compiler [29] achieves similar results through the application of predicate promotion (instruction promotion) on top of the seminal RK-algorithm [32].

The problems we had to address when implementing the LAO IF-conversion are (1) the identification of the control region to predicate, (2) the correct assignment of predicates to the conditional branches exiting the predicated region (these are not removed by IF-conversion), and (3) the correct processing of the predicated code supplied as input to the IF-conversion. The second problem is related to the fact that the ST120 branches are active on the guard value false. The third problem arises when processing LAI code that contains already predicated operations.

The LAO heuristic for choosing the region for predication calculates a threshold such that when the shortest execution path is penalized beyond this threshold, the IF-conversion is not authorized. In order to address the other two issues, we modified the steps two and three of Fang’s algorithm.

The pseudo-code in figure 4 implements the modifications required to assign the new predicates \( G_p \) to the operations of each basic block in the predicated region. When an operation is previously predicated with \( G_s \), which is locally defined in the block, but has not been locally renamed (meaning that this is the first use of \( G_s \) as a predicate seen so far), function localOptimize is called to generate better code than the default code supplied to insertOperation in figure 4.

Function localOptimize first identifies the Guard Modification Instruction (GMI) that defines the predicate operand \( G_s \) of a predicated operation, then applies one of the rewrite rules listed in figure 5, where LGD means Local Guard Definition, and DCE marks likely candidates for Dead Code Elimination. In case of a predicated GOTO operation (conditional exit branch), the first rule in figure 5 applies. The second and the third rules apply to the operation patterns commonly found in already predicated LAI code.

Finally, we reuse function localOptimize in the third step of Fang’s algorithm, in order to optimize the code that computes the control predicates of the successors of the current basic block. Assuming the current basic block predicate is \( G_p \), Fang’s algorithm requires to generate code that computes \( G_s \). Fang’s algorithm maintains code that computes \( G_s \) which are predicated with \( G_p \). In case \( G_s \) is locally defined in the current basic block, one of the four last patterns in figure 5 applies.

### 3.4 The LAO Instruction Scheduler

The LAO instruction scheduler performs two main functions: before register allocation (prepass scheduling), modulo scheduling software pipelining is applied to all the inner hardware loops whose body is a superblock, while block scheduling is applied to all the other superblocks; after register allocation (postpass scheduling), block scheduling is applied to all the previously scheduled superblocks that have been modified due to spill code insertion. Superblocks refer to single-entry control flow-regions, whose basic blocks have one predecessor, and zero or one successor in the region [28].

The LAO assumes that the order found in the LAI code is significant, with the most frequent paths already straightforward. Accordingly, all the LAO phases try to maintain this order, so the superblocks are selected greedily from the incoming sequence of basic blocks. Unlike [21], we do not currently apply tail duplication to build larger superblock, as this increases the code size. Since the LAO instruction scheduler runs after the IF-conversion, its superblocks actually contain predicated code like the hyperblocks of [29].

The main issues we had to address when implementing the LAO instruction scheduler are (1) dealing with the blocking DU loads of the ST120 decoupled implementation, (2) dealing with the zero-latency dependences, (3) software pipelin-
ing of the hardware loops, in particular with regards to mod-
ulo expansion [26], and (4) branch shadow exploitation.

To address the blocking DU load problem, the LAO in-
struction scheduler tries to schedule these operations as close
as possible to their first use, without enforcing a hard schedul-
ing constraint. In order to achieve this, we used the lifetime-
sensitive scheduling framework that minimizes the MinLife
[8] of the instruction schedule. This lifetime-sensitive frame-
work, as well as the related MinBuffer framework [30], have
already been shown to be effective at lowering the register
pressure in modulo scheduling software pipelining [14].

The best part however is that these frameworks lead to
a simple and efficient network flows formulation over an ex-
tension of the dependence graph [10]. In particular, we ad-
dressed the blocking DU loads problem in the MinLife sim-
ply by increasing fourfold the weight of the register lifetimes
that originate from these loads. As a result, virtually all the
DU loads end up being scheduled by the LAO instruction
scheduler zero or one cycle away from their first use.

Precise exploitation of the zero-latency dependences on
the ST120 is important, as it reduces the register pressure,
and also the code size especially in the SLIW mode. In a
superscalar processor instruction scheduler, all the operations
that are scheduled together at any given cycle, called the is-
sue group, are assumed independent. Later in the code gen-
erator, when the instruction schedule is linearized back to a
sequence of instructions, reordering is applied inside the is-
sue groups to ensure that the processor run-time instruction
issue logic will actually recreate the same instruction sched-
ule (issue dates, functional resource bindings) as planned
by the instruction scheduler. Such issue group reordering is
always legal, as it contains independent operations.

With some zero-latency dependences, the task of the in-
struction scheduler becomes more complex, as the opera-
tions inside an issue group are no longer independent. To
solve this problem, we first use the fact that the ST120 run-
time binding of reservation tables to an instruction only
depends on the memory position ("slot") of that instruc-
tion inside an aligned instruction pair (GP32) or quadru-
ple (SLIW). The LAO instruction scheduler then exploits a
ST120 machine modelization where the reservation tables
contain extra resources to prevent the occurrence of issue
groups that could violate zero-latency dependences. For in-
stance, these reservations tables enable the (DU-LOAD, DU-
STORE) aligned instruction pair, but forbid the equivalent
(DU-STORE, DU-LOAD) instruction pair. Last, schedule
linearization emits the issue groups in memory order.

As a result, the LAO instruction scheduler can exploit the
ST120 zero-latency dependences by putting dependent oper-
ations into the same issue group, and be assured that schedule
linearization will not violate these dependences.

The LAO software pipeliner is based on the Cray T3E
modulo scheduling software pipeliner, that uses the Inser-
tion Scheduling heuristic [9], the MinLife lifetime-sensitive
scheduling framework [10], and a generalized software pipeline
construction scheme that uniformly applies to the FOR and
WHILE inner superblock loops [11]. In the LAO, software
pipelining is only applied to the inner hardware loops, since
otherwise the loop branching overhead dominates.

Because the ST120 architecture does not include rotating
register files, the software pipeliner relies on modulo expan-
sion [26] to remove WAR and WAW register dependences.

The other alternative to modulo expansion, explicit moves
from register to register in order to keep the register lifetimes
shorter than the loop iteration interval, does not apply to
the long register RAW latencies that result from moving
data or guard values from the DU to the AU/GU. Fortu-
nately, such long latencies mostly appear in scalar code, as
the full digital signal processing loops have long lifetimes
addressing patterns that can be supported by the AU alone.

A first concern with modulo expansion is the code size
increase implied by kernel renaming and the epilog blocks.
Another problem is that modulo expansion implies kernel
unrolling, which naturally yields a software pipelined loop
whose body contains multiple exits. However, multiple exit
loops defeat the purpose of the ST120 counted hardware
loops, where there is a single implicit exit at the end of the
loop body. On such counted loops, the alternative is not to
insert the early exits, and to create a remainder loop.

Therefore, the LAO software pipeliner disables modulo
renaming by default, unless explicitly overridden. Thanks
to the decoupled implementation of the ST120 and its zero-
latency DU loads, such disabling does not significantly im-
pact the performance of the digital signal processing loops.

Finally, the LAO instruction scheduler is in charge of the
branch shadow exploitation. To do so, we first apply a sim-
plesized form of predication, where the non-predicated op-
erations that must follow the branch are predicated with the
same guard as the branch. For these operations, the con-
control dependence from the branch is replaced by a data de-
pendence from the GM1 that produces this guard. Then during
instruction scheduling, a priority function on the candidate
issue slots penalizes scheduling inside the branch shadow for
the operations that would stall branch shadow execution.

3.5 The LAO Register Allocator

The LAO register allocator is designed (1) to effectively
allocate registers, (2) to reduce the number of MOVE op-
erations generated by the SSA translation at the control flow
and the predicate merge points (§3.2), and (3) to efficiently
handle spilling in the user predicated code while not having
the full information on the predicate relationships.

The LAO register allocator is effective, thanks to the im-
plementation of the graph coloring register allocation tech-
nique [6], and the provision of a Predicate Query System
(PQS) based on the Predicate Partition Graph (PPG) [25].
The LAO register allocator includes optimistic register col-
oring [3], rematerialization of constants [4], iterated register
coalescing [19], interference graph splitting by resource type
data, address, and control registers on the ST120), and im-
plements the local spill optimizations of [7].

To reduce the number of MOVE operations, the LAO reg-
ister allocator implements repeated coalescing, which intro-
duces two improvements over the iterated register coalescing
of George & Appel [19]: it incrementally updates the inter-
ference graph during coalescing, and it extends the iterated
coalescing allocator automaton in order to take advantage of
the more accurate interference information.

During the graph coloring register allocation, a register to
register MOVE operation can be removed only if its source
and destination live ranges do not interfere, i.e. there is
no edge in the interference graph connecting these two live
ranges. In such cases, existing graph coloring register allo-

a1 := def
a2 := a1 + 3
if (!cond) goto L2

L1:
a3 := a2 + b
a1 := a3
a2 := a3
if (cond) goto L1

L2: return a1

Figure 6: Live range interferences.

(a)

(b)

Figure 7: LAO speedups on the LCC code.

Figure 8: LAO speedups on the CCC code.

4. EXPERIMENTAL RESULTS

4.1 LAO on C Benchmark Codes

In this section, we present the results of experiments where the LAO is applied on a C benchmark that includes: some basic digital signal processing kernels; integer Discrete Cosine Transform (DCT) variants; searching, sorting, and string searching algorithms; and some predication-intensive codes like the SGML attribute white space normalization (strtrim.1), and the UNIX utility wc (strwc.1).

Effective exploitation of the ST120 architecture by the LAO is first demonstrated on a retargeted version of the LCC compiler [18]. In figure 7, the GP32 assembly code produced by LCC is compared to the LCC output using virtual registers, and further optimized by the LAO. The best speedups obtained are above 6, while the geometric mean speedup is 2.45. These large speedups are explained in part by the hardware loop mapping and the IF-conversion, as these are not performed by the LCC GP32 code generator.

We then experimented with the best ST120 Commercial C Compiler (CCC) available. In figure 8, we compare CCC with all its ST120 specific optimizations enabled, to CCC without the ST120 specific optimizations but further optimized by the LAO. The CCC ST120 specific optimizations include IF-conversion, hardware loop mapping, and generation of auto-modification addressing modes. Here again,
the speedups obtained are significant with a 1.28 geometric average. The LAO effects on code size are a 1.05 geometric average increase (figure 9). This code size expansion remains reasonable, given the performance improvement.

We also measured the efficiency of the LAO in extracting the SLIW instruction level parallelism. In SLIW mode, up to four GP32 instructions may execute every cycle, compared to the two-way superscalar GP32 execution. In figure 10, we compare the performance of the CCC improved by LAO with an option that maps inner loops to SLIW mode, to the same CCC improved by LAO that generates GP32 code by default. The geometric average speedup of SLIW on our benchmarks is 1.11, with 20% – 57% speedups on DSP loops, and small performance variations otherwise. The slowdowns can be explained by the more constraining SLIW slotting rules and the GP32/SLIW mode switching overhead. Further performance improvements of SLIW mode are expected when the LAO implements memory access pairing to reduce the memory bank conflicts [38].

4.2 Effects of the LAO Optimizations

In this section, we present experimental figures that focus on the two most specific features of the LAO implementation. The first feature is the LAO IF-conversion, that includes a specific region selection scheme, and a modification of the predication algorithm of Fang (§3.3). The second feature is the repeated register coalescing improvement of the iterated register coalescing of George & Appel (§3.5).

The effectiveness of the LAO IF-conversion (LIC) is demonstrated by applying it on two kinds of input (1) LCC generated, non IF-converted code (NIC), (2) CCC generated code, where simple structure-based IF-conversion (SIC) of IF-ELSE statements is performed. The SIC cases are especially interesting, as they stress the capability of the LAO to optimize already predicated code supplied as input.

Figure 11 displays the speedups obtained when LIC applied to NIC is compared to NIC alone, using a set of kernels where our region selection heuristic is applied. The LAO IF-conversion (LIC) achieves a geometric mean speedup of 1.51, with a maximum of 3.31 for strwc.1. These results clearly demonstrate the importance of the IF-conversion to successfully exploit the ST120 instruction-level parallelism.

Figure 12 displays the speedups obtained when LIC applied to SIC is compared to SIC. Here, the LAO IF-conversion achieves a geometric mean speedup of 1.72, with a maximum of 2.33 for strwc.1. This demonstrates that the LAO IF-conversion of regions significantly improves performance over the simple IF-ELSE conversion, and that the management of already predicated code by the LAO is very effective.

Due to the conservative nature of the LAO IF-conversion region selection heuristic, none of the kernels are degraded by the optimization. Although we believe that most of the opportunities for profitable IF-conversion are exploited by the current region selection scheme, we still have to compare it to profile guided region selection. However, the code expansion induced by a generalized region selection, that implies tail duplication, may limit the overall benefit.

The behavior of the repeated coalescing allocator was mea-
Table 1: Repeated Coalescing compared to Iterated Coalescing on dynamic instruction counts.

<table>
<thead>
<tr>
<th></th>
<th>DU MOVEs</th>
<th>AU MOVEs</th>
<th>All MOVEs</th>
<th>All SPILLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterated Coalescing</td>
<td>56183</td>
<td>23109</td>
<td>59292</td>
<td>21546</td>
</tr>
<tr>
<td>Repeated Coalescing</td>
<td>29659</td>
<td>21074</td>
<td>50733</td>
<td>21074</td>
</tr>
<tr>
<td>Relative Variation</td>
<td>-18.0%</td>
<td>-8.8%</td>
<td>-14.4%</td>
<td>-0.2%</td>
</tr>
</tbody>
</table>

Figure 13: LAO speedups on application codes.

Figure 14: LAO expansions on application codes.

4.3 LAO on Application Codes

The ST120 core is primarily targeted at portable and consumer applications, such as digital cellular phones, ADSL modems, hard disk drive control, etc. In this section, we focus on the LAO results on the following applications:

cxmodem Software modem control code.
servo Hard disk drive digital control loop.
efr 5.1.0 ETSI Enhanced Full Rate (EFR) vocoder [13].
amr 2.0.0 ETSI Adaptive Multi Rate (AMR) vocoder [13].
itu G723 ITU Dual Rate speech coder [24].

No modifications were applied to the source code of the ETSI and the ITU reference applications, except for a change in an include file to remap the basic ETSI DSP operators to the ST120 DSP intrinsic functions.

Figure 13 compares CCC under maximum optimization, to CCC improved by the LAO (CCC+LAO), and to CCC improved by the LAO with automatic SLIW mapping of the inner loops (CCC+LAO.SLIW). The best speedups are obtained on the amr 2.0.0: 1.52 for CCC+LAO, and on the itu G723: 1.77 for CCC+LAO.SLIW. Note that cxmodem and servo are not affected by the LAO SLIW mapping scheme, thus giving no improvements over GP32.

Code size (figure 14) of all the CCC+LAO cases show little variations. On the other hand, code size expansions are in the 20% – 24% range with inner loop SLIW mapping (CCC+LAO.SLIW). Such expansions are acceptable, given the performance improvements. We also expect inner loop SLIW mapping to be less expensive in code size once we implement some kind of profiling feedback in the LAO.

5. SUMMARY AND CONCLUSIONS

The ST120 core implements access-execute decoupling, an innovative predication model, and multiple instruction modes including the SLIW mode, an improved VLIW execution where four instructions issue every cycle. These features enable high-performance, low-power consumption, and compact code size, on digital signal processing applications. Because of these features however, the ST120 core presents unique challenges for the compiler code generation.

In this paper, we introduce the ST120 Linear Assembly Optimizer (LAO), a tool designed to successfully meet these challenges, and used to fill the need for automatic optimization of the ST120 assembly code. We present solutions to several compiler optimization problems: the SSA translation and the IF-conversion in presence of already predicated code; instruction scheduling for a decoupled implementation; and repeated register coalescing, our improvements to the graph-based coalescing register allocator.

The 1.35 – 1.52 (GP32) and 1.57 – 1.77 (SLIW) speedups obtained on the industry standard speech coders, when optimizing the assembly code output of the leading ST120 industry C compiler, demonstrate the efficiency of the LAO technology. We expect further improvements after these compilers are enhanced to emit Linear Assembly Input (LAI) code, instead of the basic ST120 assembly code.

6. REFERENCES


