디지털 신호 처리기를 위한 Retargetable 컴파일러 연구 (A Study on a Retargetable Compiler for Digital Signal Processors)

정 성 준, 조 정 훈, 백 윤 흥
한국과학기술원 전자전산학과 전기 및 전자공학전공
{sjjung, jhcho, ypaek}@soar.kaist.ac.kr

요 약

Retargetability는 많은 컴파일러에 있어서 중요한 설계 관심사이며, 특히 *범용 프로세서(GPP)*에 비해서 훨씬 변종이 많은 *디지털 신호 프로세서(DSP)*에 있어서 더욱 중요한 관심사이다. 이 논문에서 우리는 *Very Portable Optimizer(VPO)*라고 불리는 retargetable 코드 생성기를 최근 DSP를 target 하도록 개선한 우리의 초기 연구를 나타내고자 한다. VPO는 원래 GPP를 목표로 개발되었으나, retargetability를 만족시키기 위해 개발된 새로운 중간 표현(intermediate representation)에 의해, 비교적 짧은 시간에 일반 상용 DSP에도 성공적으로 retarget될 수 있었다. 본 retargetable 컴파일러에 관한 연구는 아직 초기단계이며, 그렇기 때문에 코드 성능에는 아직도 향상의 여지가 있다. 그럼에도 지금까지의 VPO의 개선으로 말미암아, 생산 현장에서 쓰일 수 있을 정도의 성능을 지니는 주문 생산된 컴파일러와 비교해서도 충분히 자극적인 결과를 얻을 수 있었다.

1. 서론

DSP는 멀티미디어와 통신기기 같은 최근의 내장형 시스템(embedded system)의 설계에 있어 가장 중요한 부분이 되었다. 사용의 증가와 함께, 가격과 성능을 만족시키기 위한 복잡성 역시 증가하였다. 이러한 복잡성의 증가로 어셈블리 프로그래밍은 매우 어렵게 되었고, 효율적인 DSP용 소프트웨어를 개발하는 것은 오늘날 DSP 시장에 있어서 가장 큰 도전과제가 되었다.

일반 사용자에게 있어 소프트웨어를 개발하는 방법은 고급 언어를 사용하는 것이다. 하지만, 이를 위해서는 고급 언어로 만들어진 코드를 효율적인 기계어로 바꾸어 주는 컴파일러 기법이 매우 중요하다. DSP 제작자들은 고급 언어를 위한 컴파일러를 제공하고 있지만, 이러한 컴파일러가 생성한 코드는 아직 손으로 생성한 코드에 비해 성능이 만족스럽지 못하다. 많은 시스템 설계자들이 지적하듯, 강력한 컴파일러의 부재는 내장형

시스템 개발에 있어 가장 큰 난관이 된다.

DSP를 위한 컴파일러의 한가지 중요한 요구조건은 바로 *retargetability*이다. 대부분의 컴파일러에도 retargetability는 중요하지만, 특히 DSP에 있어서는 더욱 중요한데, 이는 DSP의 구조와 명령어에는 매우 변종이 많기 때문이다. 이상적으로 retargetable 컴파일러는 반드시 *user-retargetable* 해야 한다. 일반적인 user-retargetable 컴파일러는 사용자가 프로세서를 기술하는 것으로 retarget 하는 것이다. 컴파일러를 직접 고칠 수도 있지만, 대략 한달 이내에 완료할 수 있을 정도로 최소화되어야 한다.

이 논문에서 우리는 *Zephyr compiler infrastructure*[2]에 기반한 user-retargetable 컴파일러에 관한 최근의 연구를 보이고자 한다. Retargetability는 Zephyr 프로젝트에 있어서 중요한 설계 목표였다. Zephyr는 다양한 GPP에 대해 retarget되어진 반면, DSP에도 가능한지는 이제껏 검증되지 않았다. 이에 우리는 *Very Portable Optimizer (VPO)* 라고 불리는 Zephyr의 코드 생성기를 개선함으로써, Zephyr를 상용 고정소수점 DSP에 retarget하였다. 이 연구를 통해, 우선 Zephyr의 중간표현과 내부 구조만을 이해한다면 비교적 적은 노력으로 Zephyr를 원하는 DSP로 retarget 할 수 있음을 밝혀내었다. 실제로, 모든 코드 생성기는 한사람이 한달 이내로 구현하였다.

이 논문은 다음과 같이 구성되어 있다. 2장에서는 VPO의 코드 생성과 최적화 과정을 설명한다. 3장에서는 DSP용 코드를 생성하기 위해 어떻게 VPO를 고쳤는지 나타낸다. 4장에서 최근 실험 결과를 제시한 뒤, 5장에서는 retargetable DSP 컴파일러와 DSP의 특성에 맞는 코드 생성에 관한 관련 연구를 소개한다. 마지막으로 6장에서는 현재 진행중인 연구와 함께 결론을 맺는다.

2. THE VERY PORTABLE OPTIMIZER

Zephyr의 코드 생성기인 VPO는 다른 몇가지 모듈과 함께 Zephyr 내부에 구현되어 있다. 이러한 모듈들은 그림 1에서와 같이 크게 3부분으로 구성되어진다.

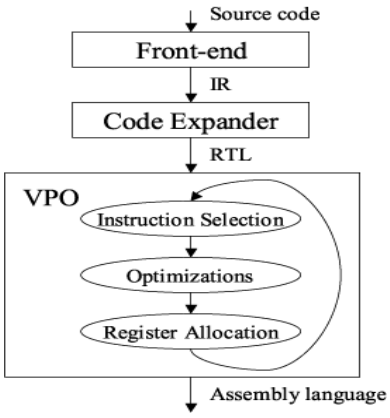

그림 1 : Zephyr의 구조

National Compiler Infrastructure[5]의 일부인 Zephyr의 설계 목표는 적은 개발비용으로 많은 다른 프로세서들을 위한 코드 생성기를 만드는데 사용되어질 수 있는 retargetable 컴파일러 제작 기반을 제공하는 것이다. 이러한 retargetability 목적을 달성하기 위해서, Zephyr의 내부에는 VPO라고 불리는 핵심 모듈이 있다. 다른 대부분의 백엔드 최적화기와는 다른 VPO만의 특징이라고 하면 모든 최적화 과정은 기계어 수준의 명령어 형태[2]에 대해서 이루어짐에도 *기계비의존적(machine-independent)*이라는 것이다. 이러한 특징은 약간의 성능 면에서의 손실을 감수함으로써 대부분의 VPO에 있는

변환을 재사용할 수 있게 해준다. Zephyr의 이런 retargetability는 *Register Transfer Lists (RTL)*[3]라고 불리는 저급의 (low-level) 중간 표현을 사용하는 것으로써 가능해진다.

RTL은 Zephyr의 모든 구성 요소들을 일관된 방법으로 서로 연관시켜주는 트리 형태 중간표현이며, 해당 기계 명령어의 의미를 나타낸다. VPO에 의해 처리되기 위해서, 프로그램은 반드시 제어 흐름 그래프(control-flow graph)로 나타내어져야 하고, 제어 흐름 그래프의 각 노드에는 명령어가 RTL 형태로 표현되어야 한다. 프론트 엔드(front-end)의 중간표현에서 RTL 형태의 제어 흐름 그래프로의 변환은 *code expander*에 의해 이루어진다. code expander를 쓰는데 있어 중요한 개념은 복잡한 명령어나, 어드레싱 모드들을 기본적인 RTL 연산(예를 들어 +나 -)과 어드레싱 모드들(레지스터 직접이나 레지스터 간접)로 쪼개는 방식으로 최대한 단순한 RTL로 출력하는 것이다. 이렇게 code expander를 단순하게 유지함으로써, Zephyr는 사용자들이 대상 아키텍처를 자세히 기술해야하지 않아도 되게 해준다. 이러한 의미로 Zephyr를 user-retargetable이라고 생각한다. 하지만 이러한 단순함은 code expander에서 출력되는 RTL이 단지 해당 기계에서 가능한 일부의 명령어만을 사용하는데 그치도록 한다. 즉, code expander로부터 나오는 RTL에는 해당 기계의 특징적인 중요한 명령어들이 포함되지 않을 수 있다. 이러한 단순하며, 간단한 RTL을 개선하기 위해서 VPO는 기계 비의존적인 변환들을 실행하여, 후에 효율적인 기계어 프로그램을 만든다.

3. Zephyr의 DSP로의 retargeting

기존에 Zephyr에 구현되어 있던 모든 코드 생성 알고리즘은 Intel Pentium이나 Sun SPARC같은 GPP를 위해서였다. 비록, 최근의 GPP는 많은 DSP적인 특징들을 가지지만, 그럼에도 아직 DSP에서만 발견할 수 있는 면이 많이 존재한다. 따라서 Zephyr에 DSP에 특징적인 면들을 적용하기 위해 VPO의 일부분도 변경하여야 하였고, 이 장에서는 이러한 개선 방안을 설명한다.

3.1 이종 레지스터 처리

3.1.1 레지스터 클래스

정의 1. 주어진 대상 기계 M에서, $I=\{i_1, i_2, \ldots, i_n\}$을 M에서 정의된 모든 명령어들의 집합이라고 하고, $R=\{r_1, r_2, \ldots, r_m\}$ 을 모든 레지스터의 집합이라고 하자. $i_j \in I$ 인 명령어 i_j에 대해 모든 피 연산자들의 집합을 $Op(i_j)=\{O_{j_1}, O_{j_2}, \ldots, O_{j_k}\}$이라고 한다. C_{j_i}이 어떤 피연산자 O_{j_i}, $1 \le i \le k$의 위치에 나타날 수 있는 모든 레지스터의 집합이라고 할 때, C_{j_i}가 명령어 i_j에 대해 ***레지스터 클래스****를 형성한다고 하고, 같은 클래스에 있는 레지스터들을* ***동종 (homogeneous)****이라고 정의한다.*

예를 들어, SPARC 아키텍처에는 3개의 피연산자를 가지는 $ALD\ reg_i, reg_j, reg_k$라는 명령어가 있다, 여기에서 레지스터 파일에 존재하는 32개의 모든 레지스터 ($r_0, r_1, \ldots, r_{31}$)는 어느 피연산자로도 나타날 수 있다. 이러한 경우에, 이 모든 레

지스터 집합은 ADD를 위한 하나의 레지스터 클래스를 형성한다. 다른 예로서, TI TMS320C5402에 있는 앞의 두 피연산자를 곱해서 세 번째 피연산자에 저장하는 $MPYA\ reg_i, reg_j, reg_k$ 라는 명령어를 생각해보자. TMS320C5402는 reg_i는 T 레지스터나 A 누산기(accumulator)가 되어야 하고, reg_j는 A이어야 하며, reg_k는 A이거나 B이어야만 한다. 이러한 경우에, MPYA는 3개의 레지스터 클래스를 가지게 된다. 즉, reg_i에는 {A. T}, reg_j에는 {A}, reg_k에는 {A, B}이다.

정의 2. 정의 1로부터, 우리는 i_j에 대한 서로 다른 레지스터 클래스를 모은 것을 S_j라고 하고 다음과 같이 정의한다

$$S_j = \bigcup_{l=1}^{k} C_{jl}$$

이것으로부터 S를 다음과 같이 정의한다

$$S = \bigcup_{j=1}^{n} S_j$$

여기서 S를 대상 기계 M에 대한 레지스터 클래스의 전체 집합이라고 정의한다.

위의 예제에서, ADD와 MPYA에 대한 S_j는 각각 {{ r_0, r_1, ..., r_{31}}}과, {{T}, {A}, {A, B}}이다. 일반적인 GPP는 동종이라고 일컫는데, 이는 S가 대개 모든 레지스터를 포함하는 단 하나 원소의 집합으로 구성되어 있기 때문이다. 즉 이것은 정의 1과 2에 의해 모든 레지스터가 모든 기계 명령어에 있어 똑같다는 것을 의미한다. 하지만 DSP의 경우에는 레지스터는 기계 명령어에 따라 다르게 할당되어 있고, 이는 단지 부분적으로 동종이라는 것을 의미한다. 예를 들어, TMS320C5402의 MPYA같은 하나의 명령어 조차도 T, A, AB라는 3개의 동종 레지스터 집합을 가진다는 것을 상기하자. 표 1은 TMS320C5402에서 정의된 레지스터 클래스의 전체 집합을 나타낸다. 일반적으로 그런 복잡한 레지스터 클래스를 가진 기계를 이종 아키텍처라고 정의한다.

ID	레지스터 클래스	의미하는 레지스터들
0	A	A 누산기
1	B	B 누산기
2	AB	A 또는 B
3	T	T register
4	ABT	A, B 또는 T
5	AT	A 또는 T
6	SP	스택 포인터
7	AR	어드레스 레지스터
8	ABTR	A, B, T 또는 AR
9	ABR	A, B 또는 AR

표 1 : TI TMS320C5402용 레지스터 클래스

3.1.2 레지스터 분류 알고리즘

GPP를 대상으로하는 gcc나 lcc와 마찬가지로 Zephyr 역시 명령어 선택과 레지스터 할당을 서로 다른 단계에서 행한다. 하지만 이러한 방식은 명령어와 레지스터가 서로 깊이 연관되어 있는 DSP에서는 제대로 동작하지 않고, *phase-coupling*을 통해 효율적으로 합쳐져야 한다. 그러한 한계를 극복하기 위해 코드 생성 알고리즘을 재작성할 수도 있지만, 그 방식은 너무 많은 노력이 들기 때문에, 우리는 현재 구현되어 있는 그 두 단계를 강제로 관계지을 수 있도록 VPO를 수정하였다. 그림 2에 새로운 VPO코드 생성 절차를 제시하였으며 그에 관한 설명은 다음과 같다. 추가한 부분은 강조를 해 두었다.

1. 우선 정의 2에 기반한 알고리즘에 따라 해당 기계에 대한 표 1과 같은 레지스터 클래스 표를 만든다.

```
1) Optimize() {
2)    ... Control and data analysis ...
3)    FindLoops()
4)    EstimateExecutionFrequency()
5)    repeat
6)       repeat
7)          LiveVariableAnalysisUpdate()
8)          DeadVariableElimination()
9)          if ColorLocalVariables() then
10)            InstructionSelection()
11)         endif
12)      until reached fixed point or used all registers
13)      ReduceType()
14)      LocalRegisterAllocation()
15)      FindHWloop()
16)      if changed then
17)         CommonSubexpressionElimination()
18)         LiveVariableAnalysisUpdate()
19)         DeadVariableElimination()
20)         LoopTransformations()
21)         InstructionSelection()
22)         InlineFunctions()
23)      endif
24)   until reached fixed point or used all registers
25)   ReduceFinal()
26)   LocalRegisterAllocation()
27)   repeat
28)      if changed then
29)         CommonSubexpressionElimination()
30)         LiveVariableAnalysisUpdate()
31)         DeadVariableElimination()
32)         LoopTransformations()
33)         InstructionSelection()
34)         InlineFunctions()
35)      endif
36)   until reached fixed point or used all registers
37)   ControlFlowTransformations()
38)   InsertFunctionPrologueandEpilogue()
39)   InstructionSelection()
40)   InstructionScheduling()
}
```

그림 2 : 변경된 VPO의 코드 생성/최적화 과정

2. code expander에서 RTL 명령어를 위해 가상 레지스터가 선택될 때, 각 가상 레지스터에는 레지스터 클래스 표로부터 클래스 항목이 첨부되어지는데, 이는 나중에 레지스터 할당 단계에서 그 가상 레지스터 자리에 같은 레지스터 클래스에 존재하는 임의의 어떤 물리 레지스터가 올 수 있게 하기 위함이다.

3. *지역 레지스터 할당기(local register allocator*, 그림 2의 14줄과 26줄)가 각 가상 레지스터를 물리 레지스터로 할당할 때, 그 레지스터와 같은 레지스터 클래스에 있는 것들 중에 하나를 고른다.

4. 지역 변수들과 파라메터들은 *전역 레지스터 할당기(global register allocator*, 그림 2의 9번째 줄)가 그래프 색칠 방식을 써서 남아 있는 물리 레지스터에 할당하게 된다.

그림 2의 13번째 줄의 ReduceType()은 레지스터 클래스의 *불일치(mismatch)*문제를 해결하기 위해 새로 추가되었다.

정의 3. 두 명령어 i_1과 i_2가 하나의 가상 레지스터 t를 처음에 i_1에 의해 쓰여지고, 순차적으로 i_2에 의해 읽혀지는 방식으로 공유한다고 가정하자. t에 첨부되어진 레지스터 클래스가 i_1과 i_2에 대해 각각 C_1과 C_2라고 하자. 그러면 만약 $C_1 \neq C_2$이면 C_1과 C_2는 ***불일치****라고 정의한다.*

레지스터 클래스들의 불일치 문제를 설명하기 위하여, 그림 3(a)의 두 수를 곱하는 수식을 생각하자. code expander가 이 수식을 RTL로 변화시킬 때, 각 RTL 피연산자에는 그림 3(b)와 같이 적절한 레지스터 클래스 후보들이 첨부되어진다. 예에서, 로드 명령어의 대상 피연산자에는 ABTR 레지스터 클래스가, 곱셈 명령어의 소스 피연산자로는 AT 레지스터 클래스가 온다. 이 경우에, 만약 B 누산기나 AR 어드레스 레지스터가 로드 명령어에서 선택된다면, 곱셈 명령어는 실행될 수 없는데, 이는 TMS320C5402에서 B나 AR은 곱셈기로의 직접적인 데이터 패스가 없기 때문이다. 여기에서 적당한 해결책은 VPO가 레지스터를 할당할 때, 두 개의 레지스터 클래스 ABTR과 AT를 그들의 교집합 (AT = ABTR ∩ AT)으로 *축소(reduce)*시

키는 것이다. 이는 다른 오버헤드나 읽어들인 값을 옮기지 않고 두 명령어간에서 할당 가능한 레지스터는 단지 AT 클래스의 레지스터뿐이기 때문이다.

이 문제는 ReduceType 루틴에서 다루어지게 된다. i_1과 i_2는 정의 3에서의 두 명령어라고 하자. 그 루틴은 입력으로서 R_d와 R_s를 받는데 이는 각각 i_1의 대상 피연산자와 i_2의 소스 피연산자이다. 루틴은 축소된 레지스터 클래스인 $R_d \bigcap R_s$를 출력한다. 만약 출력이 공집합이라면 그 명령어에 할당할 수 있는 공통의 레지스터가 없다는 것을 나타내며, 레지스터간 이동 명령어가 추가된다.

그림 3(c)는 ReduceType이 그림 3(b)의 MUL 명령어의 ABTR 레지스터 클래스를 AT 클래스로 축소시키는 과정을 보여준다. VPO가 명령어를 선택하고, 물리 레지스터를 할당한 결과는 그림 3(d)에 표시하였다.

3.2 ZOLs의 활용

TI DSP의 특징적인 면을 활용하기 위해 우리는 몇 가지 DSP 특징적인 코드 개선 기법들을 VPO에 구현하였다. 그중 중요한 하나의 기법은 *ZOLs의 활용*이다[4]. ZOLs을 지원해야하는 이유는 불필요한 분기 오버헤드를 감소시키는 것인데, 이러한 오버헤드는 TMS320C5402 같이 파이프라인 구조로 되어 있는 프로세서의 전반적인 성능을 크게 감소시킬 수 있다.

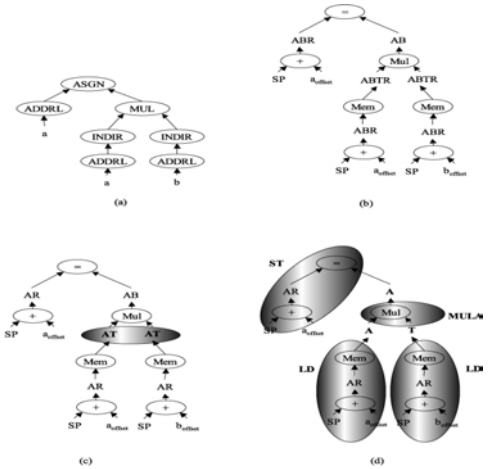

그림 3 : 이종 레지스터를 다루는 예

ZOL을 위한 코드 생성 과정은 기본적으로 [4]에서 제시된 방법과 비슷하다. 우선 VPO는 루프를 찾는다(그림 2의 3번째 줄). 그리고 발견된 루프로부터 루프 인덱스와 초기값, 한계치에 관한 정보를 모은다. 이러한 정보를 사용하여, ZOL를 제어하기 위한 레지스터를 초기화하는 RTL들이 생성된다. 그리고, 원래 루프에 있는 비교와 분기를 위해 사용되던 RTL들이 제거되게 된다.

ZOL을 어떻게 찾는지 보여주기 위해, 그림 4(a)에 있는 루프로부터 생성된 그림 4(b)의 RTL들을 생각해보자. RTL에 있는 각 레지스터들은 두 수에 의해서 표기되는데, 이는 각각 레지스터 클래스 ID(표 1 참조)와 해당 클래스에서의 레지스터 번호를 나타낸다. 이것은 3.1.2장에서 언급하였듯이, code expander가 RTL의 피연산자에 레지스터 클래스에 관한 정보를 붙인 것이다.

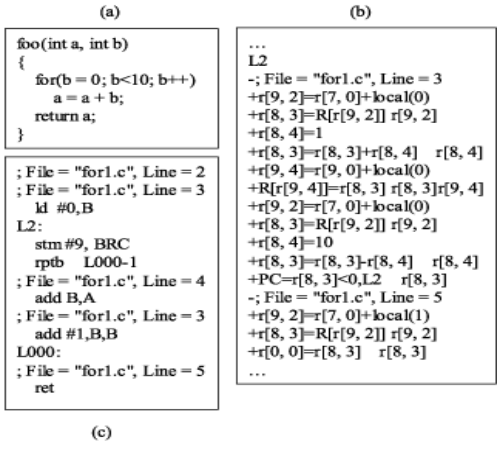

그림 4 : ZOL을 찾는 예

그림 4(b)에서는 ZOL을 사용하지 않는 RTL이며, 이것은 그림 4(c)로 변하게 된다. 그림 4(c)의 7번째 줄에서 10번째 줄까지의 명령어 구간은 10번 반복되게 되며, 반복회수는 그림 4(c)의 5번째 줄에 의해 설정된다. 반복할 구간은 그림 4(c)의 6번째 줄에 의해 정의되며, 후에 구간 반복 ZOL로 변환된다.

4. 실험

우리는 이렇게 개선된 Zephyr를 최근 TMS320C5402상에서 검증하였다. 이 장에서 이러한 최근의 실험 결과를 보여주고, 각 컴파일러 기법의 영향을 성능을 통해 분석한다. 또한 고유 컴파일러의 성능을 따라잡기 위해 VPO에 추가한 몇 가지 기법에 대한 검증의 일환으로 이러한 우리의 구현 결과를 TI 고유 컴파일러의 결과와 비교하였다.

4.1 컴파일러 기법의 효율성

TI TMS320C5402에서 수행될 때 성능을 알아보기 위해서 DSPStone 벤치마크[1]를 사용하였다. DSPStone은 일반적인 DSP 알고리즘을 수행하는 작은 C 커널들의 집합이다. 비록 크기는 작지만, DSPStone의 결과는 그러한 작은 부분 코드들로 구성된 실제 프로그램들에 대해 컴파일러가 생성한 코드의 성능을 비교적 정확히 예측할 수 있다.

벤치마크에 대한 각 기법들의 성능을 분리하기 위하여, 우리는 여러 전략을 사용해서 실험을 하였다. 각 전략들은 차츰 좋은 성능을 얻기 위해 다음과 같이 기법을 하나씩 추가하는 방법으로 설계하였다.

1. (*기본*) Zephyr를 DSP에 맞추어 retarget하기 위해 반드시 필요한 3장의 레지스터 분류 알고리즘을 제외한 어떠한 최적화 기법도 사용하지 않았다.

2. (*VPO 최적화*) 이미 VPO에 구현되어 있던 최적화 기법들만을 가능하게 하였다. 하지만 DSP 특징적인 기법들은 이 단계에서는 사용되지 않았다.

3. (*MAC 가능화*) VPO가 DSP 특징적인 MAC 명령어를 찾을 수 있도록 전략 2를 확장하였다. MAC 명령어에 대한 기계 기술을 추가하였고, VPO에 있는 코드 합치기 과정에서 그 명령어를 선택하도록 하였다.

4. (*ZOL 가능*) 전략 3이 3.2장에서 제시된 ZOL을 찾아서 사용하는 방식을 통해 확장되었다.

5. (*AR 포함*) 전략 4에서 우리는 레지스터 분류 알고리즘에 있어 단지 A와 B, T의 데이터 레지스터만을 사용하였다. 전략 5는 여기에 어드레스 레지스터를 추가하였다.

6. (*Spill 제거*) 복잡한 이종 레지스터를 다루기 위해, 레지스터 할당기가 반복적으로 호출된다. 이러한 과정에서, 많은 불필요한 spill 코드가 생성되었다. 그래서 그것들을 없애기 위해 단순한 알고리즘을 적용하였다.

다른 전략에 따른 각 실험에서, 벤치마크에 대한 VPO 출력코드의 수행 시간을 얻어서 TI 컴파일러가 최적화 레벨 3(최고)으로 생성한 출력 코드의 수행시간으로 표준화하였다. 이때, TI 컴파일러의 성능을 1로 하였다. 그림 5은 결과를 보여준다. 우리가 결과로부터 알 수 있는 한가지는 전략 2는 항상 전략 1보다 성능이 낫다는 것이다. 이는 이미 VPO에 구현되어 있는 최적화 기법이 기계 비의존적이며, DSP든 GPP든 어떤 기계에 있어서도 효율적일 것이므로 이러한 결과

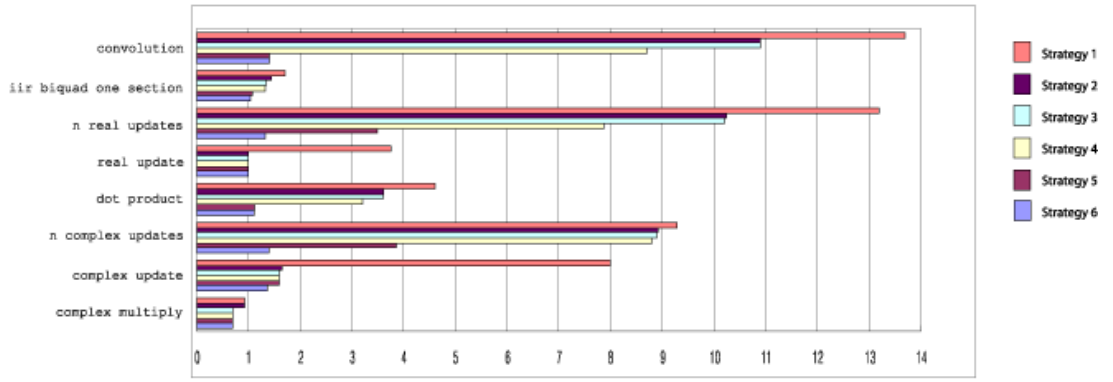

그림 5 : DSPStone 벤치마크에 대한 VPO의 TI 결과 대비 표준화 된 실행결과

는 어느 정도 짐작할 수 있는 것이었다. 또한 전략 4는 많은 경우에 있어서 전략 3보다 성능이 나은데, 이는 루프만 있다면 ZOL을 사용하는 것이 효과적이기 때문이다.

하지만, ZOL을 찾는 기법이 효과적인 것과는 달리, 많은 벤치마크들이 MAC 연산을 가지고 있음에도, 전략 3은 전략 2에 비해 성능에 있어 많이 다르지 않다. 이 결과는 AR이 전략 3에서는 아직 지원되지 않기 때문으로 설명될 수 있다. 벤치마크의 대다수는 곱셈과 덧셈 연산 사이에 포인터 연산을 많이 수행한다. 포인터 연산은 필요 불가결하게 주소 계산 코드들을 사이사이에 삽입하게 된다. 그러므로 컴파일러가 AR을 어드레스 연산에 사용함으로서 그러한 코드들을 없애지 못한다면, 곱셈과 덧셈 연산은 하나의 MAC으로 합쳐질 수 없다. 이것을 묘사하기 위해서 그림 6의 convolution이라는 벤치마크의 일부를 생각해보자. 그림 6(b)는 그림 6(a)에 있는 원래 C 코드로부터 전략 3에 의해 생성된 루프 본체에 대한 RTL을 보여준다. 여기서 주소 계산을 위해 로드와 스토어를 위한 많은 RTL들이 삽입되었음을 알 수 있다. 그러한 메모리 연산들은 VPO가 포인터 연산들로부터 자동 증가/감소 어드레싱 모드를 찾아내어서 일련의 RTL들을 하나의 MAC 연산으로 인식하는 것을 막는다.

이러한 대부분의 명령어들은 AR과 함께 자동 증가/감소 어드레싱 모드가 적절히 사용되어서 그림 6(c)에서 보듯이 불필요한 주소 계산과 메모리 연산만 없어진다면 제거되어질 수 있다. 이 개선은 또한 A(즉 r[0,0])를 이용하는 ALU 연산을 제거하며, 이것은 그림 6(d)에서 보듯이 A를 지역변수(y)에 할당함으로써 코드 성능을 더욱 향상시킬 수 있다. 결과로, 남아있는 모든 3개의 명령어들은 쉽게 그림 6(e)에서 보이는 하나의 RTL로 합쳐질 수 있게 된다. 이러한 결과는 후에 다음 기계 명령어로 변하게 된다

MAC *AR3, *AR2, A, A

```
convolution.c
for (i = 0; i < LENGTH; ++i)
y += *px++ * *ph--
```

(a)

```
r[0,0]=R[r[7,0]+px];
R[r[7,0]+l0_1]=A;
R[r[7,0]+px]=R[r[7,0]+px]+1;
r[0,0]=R[r[7,0]+ph];
R[r[7,0]+l0_2]=A;
R[r[7,0]+ph]=R[r[7,0]+ph]+1;
r[6,3]=r[0,0];
r[0,0]=R[r[7,0]+l0_1];
r[6,2]=r[0,0];
r[3,0]=R[r[6,3]];
r[1,0]=R[r[6,2]]*r[3,0];
r[0,0]=R[r[7,0]+y];
r[0,0]=r[0,0]+r[1,0];
R[r[7,0]+y]=r[0,0];
```

(b)

```
r[3,0]=R[r[6,3]d];
r[1,0]=R[r[6,2]i]*r[3,0];
r[0,0]=R[r[7,0]+y];
r[0,0]=r[0,0]+r[1,0];
R[r[7,0]+y]=r[0,0];
```

(c)

```
r[3,0]=R[r[6,3]d];
r[1,0]=R[r[6,2]i]*r[3,0];
r[0,0]=r[0,0]+r[1,0];
```

(d)

```
r[0,0]=r[0,0]+(R[r[6,2]i]*R[r[6,3]d]);
```

(e)

그림 6 : convolution 벤치마크의 코드 합침 과정

전략 5에서 이러한 보든 코드 최적화는 단지 이전 전략보다 더 좋은 성능을 내기 위해 AR을 적용하는 것으로 이루어졌다. 하지만 아직도 두 개의 벤치마크, n complex update와 n real update에 대해서는 우리의 실험 결과가 TI 것보다 훨씬 느리다. 이것은 주로 우리의 레지스터 할당기에 의해 생성된 불필요한 spill 코드 때문이다. 이 문제는 단지 새로운 VPO의 구현에서만 발생하는데, 여기서 이종 레지스터 아키텍처를 다루기 위해서 지역 레지스터 할당기(그림 2의 14번째 줄)를 반복적으로 호출하도록 설계하였기 때문이다. 하지만 우리는 이러한 불필요한 spill 코드들을 비교적 단순한 방식으로 제거할 수 있다는 것을 알게 되었고, 이것은 전략 6에서 전반적인 실행 시간을 감소시켜준다.

전략 6에서 Zephyr는 전반적으로 TI 컴파일러와 거의 동등한 성능을 보여준다. 심지어 complex multiply같은 벤치마크에서는 우리의 코드가 TI에 비해 거의 2배나 빠르다. 주된 이유는 우리는 직접 어드레싱 모드를 사용하였기 때문이다. 반면 TI는 간접 어드레싱 모드를 사용했고, AR을 초기화하기 위한 부가적인 명령어들을 생성하였는데, 우리는 그럴 필요가 없었기 때문이다.

5. 관련 연구

DSP를 위한 retargetable 컴파일러의 개발에 있어서 이에 관한 몇 가지의 연구가 있다. 이 장에서는 그 중에서 3가지의 잘 알려진 컴파일러에 대해 설명한다.

5.1 SPAM

SPAM은 Princeton에서 개발한 고정소수점 DSP를 위한 retargetable 컴파일러이다. SPAM은 Stanford University Intermediate Format(SUIF) 컴파일러를 C 프론트엔드로 해서 C 코드를 SPAM을 위한 중간 표현으로 변화시킨다. 이어서 SPAM은 코드 밀도를 높이기 위한 일련의 기계 비의존적인 최적화를 수행한다. 마지막으로 SPAM의 백엔드인 TWIF은 기계 의존적인 최적화를 수행한다. TI TMS320C25의 경우 자동변수들에 대한 그래프 색칠, 정적 할당 변수에 대한 메모리 도안, clique-covering기반의 지역 압축 같은 최적화를 수행한다. SPAM의 최적화는 단일 패스이며, TWIF은 동적 프로그래밍에 기반한 수정 Aho-Johnson 알고리즘을 사용하여 한번의 하향식(top-down) 패스로 scheduling을 한다.

5.2 RECORD

RECORD는 고정소수점 DSP를 위한 retargetable 컴파일러이다. 목표 프로세서는 MIMOLA HDL로 기술되고, 소스 프로그램은 DSP 알고리즘의 특별한 면을 나타내기 위해 데이터 흐름 언어로 주어진다. RECORD는 이러한 데이터 흐름 언어로 주어지는 소스 프로그램을 내부 제어/데이터 흐름 그래프 표현식으로 나타낸다. 프로세서 특징적인 코드 선택기는 이러한 트리로부터 차례로 명령어 선택과 레지스터 할당을 한다. 이후 scheduling 단계와 주소 할당 단계를 거친 뒤, 최종적으로 압축 단계를 통과하게 된다. 결과는 어셈블리 코드가 아닌 이진 기계 프로그램이다.

RECORD는 목표기계의 행동적이나 구조적인 기술만으로 retarget 될 수 있다. 모델로부터, 명령어 집합이 추출되고, 이 명령어 집합 추출은 코드 생성시 불필요한 구조적인

면들을 감추며, 프로세서 특징적인 코드 선택기를 생성한다.

5.3 LANCE

LANCE는 retargetable 컴파일러를 위한 인프라이다. C 프로그램을 입력으로 받아들여서, 저급의 어셈블리와 비슷한 C 문법으로 되어 있는 중간 표현을 생성한다. 그러므로 언제든 그 중간 표현은 원래 C 프로그램과 마찬가지로 컴파일 되어 목적 코드를 생성할 수 있다. 순차적으로 일련의 기계 비의존적인 최적화가 이 중간 표현에 대해 여러 가지 코드 개선 기법들을 적용한다. 코드 생성은 OLIVE와 IBURG같은 잘 알려진 트리 패턴 매처 생성기를 통해서 이루어진다. 이러한 백엔드 도구와 호환가능하기 위해서 LANCE는 자신의 three address code 중간 표현을 데이터 흐름 트리로 바꾸기 위한 도구를 제공해준다.

6. 결론과 향후 과제

이 연구의 공헌은 Zephyr라는 GPP를 위해 개발된 컴파일러를 확장하여 DSP라는 불규칙적인 아키텍처로 retarget하는 가능성을 보여준 것이다. 이 논문에서, 우리는 우리의 컴파일러가 상업용의 주문 생산된 컴파일러와 비교 가능한 성능을 제공한다는 초기 실험 결과를 제시하였다. 이러한 결과로부터 컴파일러가 Zephyr와 같은 retargetability를 제공하면서도, 대표적인 표준 DSP에 성공적으로 retarget 되었다는 것을 알 수 있다.

아직도 재미있는 주제들이 향후 과제로서 많이 남아있다. 예를 들어 [4]에서는 최근 ZOL과 그 버퍼를 활용하는 몇 가지 방안에 대해 발표하였다. 비록 몇 가지 기법들은 DSP의 특징적인 면에만 적용 가능하지만, loop collapsing이나 loop interchange 같은 기법들은 TMS320C54x에도 구현하는 것이 도움이 된다는 것을 알아내었고, 따라서 본 논문에서 제시한 우리의 결과보다도 더욱 나은 성능을 얻을 수 있다는 것을 가르쳐준다.

참고문헌

[1] V. Zivoljnovic, J.M. Velarde, C. Schager, and H. Meyr. DSPStone - A DSP oriented Benchmarking Methodology. In Proc. of International Conference on Signal Processing Applications and Technology, 1994.

[2] A. Appel, J. Davidson, and N. Ramsey. The Zephyr Compiler Infrastructure. Technical Report at http://www.cs.virgina.edu/zephyr, University of Virginia, 1998.

[3] N. Ramsey and J. Davidson. Machine Descriptions to Build Tools for Embedded Systems. In Workshop on Languages, Compilers and Tools for Embedded Systems, 1998.

[4] G. Uh, Y. Wang, D. Whalley, S. Jinturkar, Y. Paek, C. Burns, and V. Cao. Compiler Transformation Techniques for Effectively Exploiting a Zero Overhead Loop Buffer. submitted to Software-Practice and Experience, 2001.

[5] J. Davidson. NCI: National Compiler Insfrastructure Overview. In Proc. of The National Compiler Infrastructure Tutorial In conjunction with the ACM Conference on Programming Language Design and Implementation, Vancouver B.C., Canada, June, 1998.