모듈진화를 이용한 효율적인 진화 하드웨어 설계
(An Effective Evolvable Hardware Design using Module Evolution)

황 금 성 † 조 성 배 ††
(Keum-Sung Hwang) (Sung-Bae Cho)

요 약 진화 하드웨어(Evolvable Hardware)는 환경에 적응하여 스스로 구성을 변경할 수 있는 하드웨어로 생산성 향상 및 독창적 회로설계를 위해 최근 널리 연구되고 있다. 하지만, 하드웨어의 복잡도가 증가할수록 진화를 위해 탐색해야 하는 해공간의 크기가 기하급수적으로 증가하기 때문에 아직까지 복잡한 하드웨어에 대해서는 좋은 활용방안을 찾지 못하고 있다. 이 논문에서는 좀더 효율적인 설계를 위하여 복잡한 하드웨어를 모듈별로 나누어 진화시키는 방법을 제시한다. 몇 가지 회로를 기존 회로 진화 설계 방식과 제시하는 모듈진화 방식으로 실험하여 비교한 결과 약 50배에서 1,000배까지의 세대절약 효과를 얻을 수 있었으며, 좀더 최적화된 하드웨어를 얻을 수 있었다.

키워드 : 복잡한 진화 하드웨어, 모듈 진화, 모듈 결합, 디지털 회로 설계, 해공간

Abstract Recently Evolvable Hardware (EHW) is widely studied to design effective hardware circuits that can reconfigure themselves according to the environment. However, it is still difficult to apply for complicated circuits because the search space increases exponentially as the complexity of hardware increases. To remedy this problem, this paper proposes a method to evolve complex hardware with a modular approach. The comparative experiments of some digital circuits with the conventional evolutionary approach indicate that the proposed method yields from 50 times to 1,000 times faster evolution and more optimized hardware.

Key words : complex evolvable hardware, modular evolution, module combination, digital circuit design, fitness landscape

1. 서 론

1990년대 FPGA(Field Programmable Gate Array)라는 프로그래밍 가능한 하드웨어가 개발된 이래로 진화 하드웨어(Evolvable Hardware)라는 분야가 관심을 모으고 있다. 진화를 이용한 하드웨어 설계란 논리 게이트로 구성된 하드웨어를 염색체 집단으로 표현하고 인공적인 진화를 통해 원하는 전자 회로의 설계를 얻어내는 방법인데, FPGA와 같이 하드웨어의 구성 상태를 실시간으로 변경 가능한 하드웨어가 필요하다. 진화 하드웨어는 실시간으로 진화가 가능하기 때문에 변화가 많고 예측하기 힘든 환경 속에서도 하드웨어를 최적의 상태로 적응하여 유지하는 기능을 기대할 수 있다.

† 학생회원 : 연세대학교 컴퓨터과학과
yellowg@candy.yonsei.ac.kr
†† 종신회원 : 연세대학교 컴퓨터과학과 교수
sbcho@csai.yonsei.ac.kr
논문접수 : 2002년 6월 12일
심사완료 : 2004년 8월 26일

이러한 진화 하드웨어의 가능성이 알려지면서 이를 실용화하기 위한 많은 연구가 이루어져 왔다. Sipper 등[1]과 Yao 등[2]의 연구를 통해 진화 회로 설계를 위한 다양한 방법이 제시되었으며, Thompson[3]은 디지털 회로의 효율적인 진화에 대한 연구들을 종합하여 소개하였다. 이러한 연구들은 기본적으로 회로 구성단위의 기능과 그 구성단위 사이의 연결 방법의 진화를 제시하고 있다. 그러나 이제까지 개발된 방법들은 진화를 통해 얻을 수 있는 하드웨어의 크기에 제한이 있다. 하드웨어가 크고 복잡할수록 이를 표현하는 데 필요한 염색체의 길이가 길어지며, 이에 따라 진화에 요구되는 시간이 기하급수적으로 늘어나기 때문에 아직 실용적으로 복잡한 회로를 진화 하드웨어로 구현하기는 어렵다. 이를 위해서는 복잡한 하드웨어를 좀더 효율적으로 진화시킬 수 있는 방법이 요구된다.

이러한 문제를 해결하기 위해서 여러 연구 및 방법들이 제시되었다. 게이트 회로를 진화의 단위로 사용하는 게이트 수준의(gate-level) 진화 하드웨어 대신, 좀더 높

은 수준의 하드웨어 기능을 가진 기능을 진화의 단위로 사용하는 기능 수준의(function-level) 진화 하드웨어 설계 방식이 Liu 등[4]에 의해 제안되었으며, 설계하고자 하는 하드웨어를 한 번에 진화시키지 않고 여러 부분으로 나누어 진화한 다음 결합하여 원래의 하드웨어를 얻고자 하는 분할정복(divide-and-conquer) 하드웨어 설계 방법이 Torresen[5]에 의해 제시되었다. 또한 Vassilev[6]는 적합도 공간을 분석하여 진화 하드웨어의 확장성을 보임으로써 모듈과 같은 좀더 큰 하드웨어 단위를 진화시키는 방법의 가능성을 입증하였다. Koza가 제안한 유전자 프로그래밍(genetic programming)을 이용한 연구에서는 하드웨어 서브모듈을 진화에 사용되는 단위 함수로 정의하여 진화적인 조합을 통해 좀 더 복잡한 하드웨어를 얻는 방법을 제안하였다[7]. 여기에서 Koza는 진화에 의해 얻어진 부분 모듈의 재사용성을 보장하기 위한 여러 방법들을 제안하며 복잡한 하드웨어를 효율적으로 진화시키는 방법에 대해 논의하였다[8]. 하지만 유전자 프로그래밍은 진화 하드웨어 상에서 작동시키기엔 복잡하고 어려운 방법으로 알려져 있어 아직까지는 직관적이고 다루기 쉬운 유전자 알고리즘(GA: genetic algorithm)이 많이 사용되고 있다.

본 논문에서는 이러한 연구의 일환으로 진화 하드웨어 상에서 복잡한 회로를 진화시키기 위하여, 하드웨어의 부분 모듈을 진화시키고 효율적으로 결합하는 방법을 제안한다. 이제까지 모듈화된 분할 진화 방법은 범용의 분할 방법이 아니라, 목표로 하는 하드웨어의 특성에 따라 하드웨어의 부분을 나누어 진화 및 결합하는 것이 보통이다. 하지만, 본 논문에서는 출력 단자를 기준으로 진화될 하드웨어의 모듈을 나누어 전문적인 지식 없이도 분할·진화 및 결합하는 방법을 제안한다. 이 방법은 입출력 패턴이 명확하고 출력의 수가 일정한 디지털 회로의 설계에서 범용적으로 사용가능하리라 기대된다.

2. 디지털 회로의 진화

디지털 회로를 진화시키기 위해서는 일반적으로 유전자 알고리즘을 사용한다[9]. 각 하드웨어 구조를 나타내는 염색체 집단을 임의의 구조로 초기화시키고, 원하는 하드웨어의 입출력 패턴과 유사한 정도를 적합도로 사용하며, 만족스러운 염색체가 발견될 때까지 유전 연산(genetic operation)을 적용하여 진화시키는 방법이다. 일반적으로 진화 알고리즘을 이용하여 하드웨어를 설계하는 과정은 그림 1과 같다.

그림 1에서 볼 수 있듯이 진화 하드웨어는 독특한 적합도 평가 과정을 거친다. 일반적으로 진화 알고리즘은 명시적 함수를 통해서 적합도를 측정하지만, 진화 하드웨어는 염색체가 표현하는 하드웨어 설계를 실제로 적

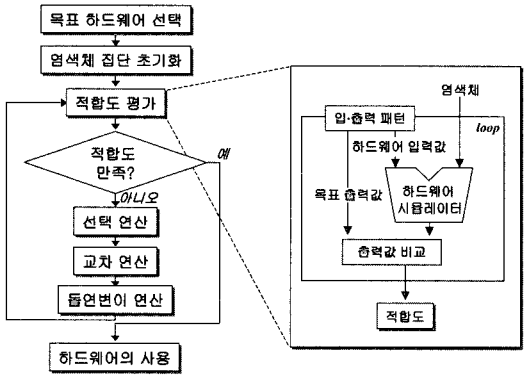


그림 1 진화 알고리즘을 이용한 하드웨어의 설계 과정

용하여 하드웨어 기능이 얼마나 잘 작동하는지를 적합도로 사용한다. 이때 작동 우수성은, 모든 가능한 입력값을 넣었을 때 나오는 출력 패턴이 원하는 결과와 일치하는 정도로 평가한다. 본 논문에서는 이 값을 퍼센트(%)로 환산하여 적합도는 0에서 100사이의 값을 적합도로 사용한다.

디지털 회로를 진화시키기 위해서 사용되는 하드웨어는 FPGA(Field Programmable Gate Array)와 같은 프로그램 가능형 논리 장치(PLD: Programmable Logic Device)이다. PLD는 사용자가 실현하고자 하는 임의의 기능을 간결하고 고속으로 실현하고자 하는 요청과 개발의 용이성을 고려하여 여러 번 전기적인 기능교체가 가능하다. 기본적인 구조는 논리마크로셀(LMC: Logic Macro Cell)과 마크로셀 사이의 연결 장치로 구성되어 있다. 논리마크로셀은 범용의 논리회로로, 기능지정 비트열에 따라 기능이 다양하게 바뀐다. 그리고, 연결장치는 이 마크로셀들을 2차원상의 휴즈 배열에 의해 임의로 상호연결해 준다. 따라서 논리마크로셀의 기능을 결정하는 비트열과 휴즈 배열의 패턴을 지정하는 비트열이 회로의 "프로그램"을 결정짓게 되며 이를 합쳐 아키텍처 비트라고 한다. EHW에서는 이 아키텍처 비트를 염색체로 간주하여 하드웨어를 진화시킨다[10].

본 논문에서 사용된 진화 하드웨어의 구조는 그림 2와 같다.

이 하드웨어는 n_I개의 입력값을 받아서 n_O개의 출력값을 내보낸다. 하드웨어 내부에는 $m \times n$개의 논리기능셀들이 $m \times n$ 배열로 위치되어 있고, 각 셀은 상호연결되어 있어 입·출력값을 주고받을 수 있다. 논리기능셀은 앞서 소개한 논리마크로셀과 같이 여러 기능을 가지고 있는 장치이며, 여기서는 10가지 종류의 게이트 회로의 기능으로 작동하도록 정의하였다. 논리기능셀에서 선택 가능한 기능은 표 1과 같으며, 각각 번호를 부여하여 기능을 구분하였다.

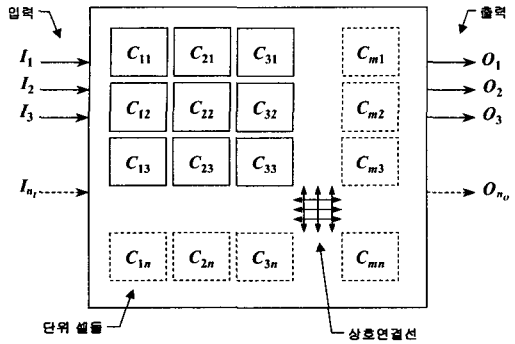


그림 2 $n \times m$ 개의 논리기능셀이 회로에 배열되어 있는 일반적 형태의 재구성 가능 하드웨어. 각 단위셀은 선택된 논리 기능으로 작동하며 상호 연결되어 회로를 구성한다.

표 1 논리기능셀에서 선택 가능한 10종류의 논리 기능. ¬는 NOT, ∧는 AND, ∨는 OR, ⊕는 XOR을 나타낸다. $(\neg x) \oplus y$와 $x \oplus (\neg y)$, $\neg(x \oplus y)$는 같은 기능이므로 하나만 사용한다.

번호	기능	번호	기능
0	$x \wedge y$	5	$x \vee (\neg y)$
1	$x \vee y$	6	$(\neg x) \vee y$
2	$x \wedge (\neg y)$	7	$(\neg x) \oplus y$
3	$(\neg x) \wedge y$	8	$\neg(x \wedge y)$
4	$x \oplus y$	9	$\neg(x \vee y)$

외부에서 들어온 입력값은 아키텍처 비트에 의해 결정된 상호연결 조합과 논리기능의 조합을 거쳐서 원하는 출력값을 내보낸다. 그림 2의 하드웨어를 구성하는 아키텍처 비트를 표현하기 위해서 각 셀에 다음과 같이 세 가지 값을 할당한다.

$$Cell: (ci_1,\ ci_2,\ f) \tag{1}$$

위에서 ci_1과 ci_2는 입력값을 전달해줄 셀을 의미하며, f는 논리기능셀의 기능을 의미한다. 각 논리기능셀은 2개의 입력을 이전 열에 위치한 셀로부터 받는다. 이때 첫 열에 위치한 셀은 외부입력으로부터 값을 받는다. 외부입력들도 각각 하나의 셀로 간주되어 논리기능셀로 값이 전달되는 것이다. 논리기능셀에 들어온 입력값은 f에 의해 선택된 논리기능이 적용된 다음 출력으로 내보내진다. 본 논문에서 사용된 f 값은 표 1과 같다. 각 셀의 입력값은 중복된 값을 받을 수 있는데, 이는 f에 의해 선택된 논리기능과 조합되어 아래와 같이 입력값을 출력값으로 그대로 내보내는 기능과 NOT을 적용하여 내보내는 기능, 그리고 항상 0이나 1이 되도록 하는 기능이 된다.

진화에 사용될 염색체는 수식 1에서 정의된 각 셀의 정보를 그림 2와 같이 배열로 연결하여 사용한다. 그리고, 염색체의 끝부분에는 하드웨어의 외부출력 값을 나타내는 셀을 선택하는 출력셀이 위치한다.

$$\forall x, \quad x \wedge x \Leftrightarrow x, \quad \neg(x \wedge x) \Leftrightarrow \neg x, \quad (\neg x) \wedge x \Leftrightarrow 0, \quad (\neg x) \vee x \Leftrightarrow 1 \tag{2}$$

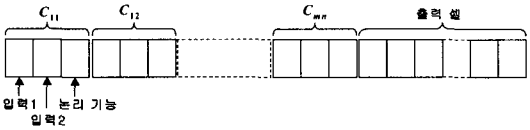


그림 3 $n \times m$ 개의 논리기능셀과 n_o개의 출력셀을 하나로 나열하여 구성한 염색체의 모습

각 셀에 들어오는 입력 셀은 약속된 번호로 표현하여, 외부입력 I_0에서부터 논리기능셀과 출력셀까지 차례로 번호를 부여하여 사용한다. 그리고 편의상 각 셀은 이전 열에 있는 셀의 출력만을 사용할 수 있도록 하기 때문에, 2번째 열에 위치한 셀에서 외부입력의 값을 직접 사용하기 위해서는 첫 번째 열에 그 값을 그냥 통과시켜 주는 셀이 필요하다.

3. 모듈진화를 이용한 진화 하드웨어

기존의 진화 하드웨어 설계 방법에서는 모든 입출력 패턴을 비교하면서 전체 하드웨어를 설계해낸다. 즉, 얻고자 하는 목표 하드웨어 기능(F)의 구성 회로를 한 번에 진화시킴으로써 기존의 설계 방법보다 최적화된 구조를 얻을 수 있는 것이다. 그러나 복잡한 하드웨어의 경우 전체 구조를 한 번에 설계하는 것은 어려운 일이고 많은 시간이 요구된다. 본 논문에서는 하드웨어를 모듈별로 분리하여 진화한 뒤 결합하는 방법을 제안한다. 사용한 연산자는 표 2와 같다.

표 2 모듈 진화를 위한 연산자

연산자 표시	의미
⊔	디지털 회로의 최적화된 결합
⊓	게이트 레벨에서 공유 가능한 회로 (입출력 패턴이 같은 회로)
⊕	디지털 회로의 단순 결합
⊖	해당 중복 기능을 제거하고 재사용 처리

표 2에서 소개한 하드웨어 결합방법에서, ⊔는 진화를 통해 궁극적으로 찾고자 하는 최적의 결합을 나타내며, ⊕는 단순히 기능을 나열하는 결합을 의미한다. 그리고 ⊖는 ⊓를 통해 찾아낸 중복된 하드웨어를 하나만 남기고 제거하여 재사용하도록 하는 연산자이다. 그림 4는 이 연산자들의 관계를 잘 나타내는 예제이다. 여기서 '='

는 동일한 회로, '⇔'는 입출력 패턴이 같은 동치 회로를 나타낸다. 그림에서 가장 오른쪽 아래에 있는 회로가 궁극적으로 찾고자 하는 목적회로이고, 1번째 행은 회로1을 나타내며 2번째 행은 회로 2를 나타낸다. 그림을 보면 가장 오른쪽의 '⊔' 연산만 거쳐도 목적회로를 찾을 수 있고, 왼쪽의 '⊕' 연산과 '⊓', '⊖' 연산을 거쳐서도 목적회로를 얻을 수 있다. 이때 '⊕', '⊓', '⊖' 연산을 이용하기 위해서 1번째 행의 오른 쪽에 위치한 회로1이 왼쪽의 회로로 변환되었다. 이는 '⊕', '⊓', '⊖' 연산자를 사용하기 위해서는 연산이 가능한 동치회로의 탐색이 먼저 요구됨을 보여주는 예이다.

표 2에서 정의한 연산자를 이용하여 하드웨어 기능(F)을 여러 개의 모듈이 결합된 형태로 나타내면 다음과 같다.

$$F = f_1 \sqcup f_2 \sqcup \cdots \sqcup f_{n_M} \tag{3}$$

수식 (3)에서 f_k (k=1, 2, ···, n_M)는 전체 하드웨어의 일부 기능을 구성하는 부분 하드웨어 혹은 모듈을 나타내고, n_M은 그 수를 나타낸다. 위 식에서 전체 하드웨어 F는 부분 기능 f_k가 모여서 구성되어 있다. 따라서 하드웨어를 결합하는 ⊔ 연산을 할 수 있다면 f_k를 통해 F를 구할 수 있다. 그러나 하드웨어 모듈을 최적으로 결합하는 일반적인 방법은 알려져 있지 않다. 따라서 ⊔ 연산을 사용하지 않고 f_k를 결합하여 F를 구하는 방법을 찾아보기로 한다. 먼저 g_k를 다음과 같이 정의한다.

$$g_k = f_1 \sqcup f_2 \sqcup \cdots \sqcup f_k \tag{4}$$

여기서 g_k와 F는 다음과 같은 관계를 가지고 있다.

$$F = g_{n_M} \tag{5}$$

식 (5)에서 g_k를 구하는 방법을 안다면 F도 구할 수 있음을 알 수 있다. g_k는 다음과 같은 귀납적인 관계를 가지고 있다.

$$\begin{aligned} g_{n_M} &= (g_{n_M-1} \oplus f_{n_M}) \ominus (g_{n_M-1} \sqcap f_{n_M}) \\ g_{n_M-1} &= (g_{n_M-2} \oplus f_{n_M-1}) \ominus (g_{n_M-2} \sqcap f_{n_M-1}) \\ &\vdots \qquad\qquad \vdots \\ g_2 &= (g_1 \oplus f_2) \ominus (g_1 \sqcap f_2) \\ g_1 &= f_1 \end{aligned} \tag{6}$$

식 (6)은 식 (4)를 ⊕, ⊖, ⊓ 연산자를 이용해서 표현한 것이다. f_k는 진화를 통해 얻을 수 있는 부분 회로이고, ⊕도 연산이 가능한 단순 결합이다. 그리고 ⊖는 제거하려는 부분이 중복되어 존재하는 회로일 경우 연산이 가능하다. 따라서 ⊓ 연산만 해결하면 식 (4)를 통해 ⊔ 연산을 할 수 있게 된다. 그러나 ⊓ 연산도 기능의 중복된 부분을 찾아내기 위해서 하드웨어 내부 구조를 모두 비교해야 하고, 표현이 가능한 여러 하드웨어 구조 중에서 어느 것을 선택할 것인지에 대한 문제를 가지고 있다. 따라서 본 논문에서는 완벽하지는 않더라도 비슷한 수준의 ⊓ 연산 방법을 제안하여 모듈(f_k)의 결합에 사용하였고, 비교적 좋은 결과를 얻을 수 있었다.

3.1 모듈별 진화 프로세스

모듈별 진화를 이용한 하드웨어 설계 과정은 그림 5와 같다. 기존의 방법과 다른 점은 출력 부분별로 따로 진화시킨 다음 결합하는 것이다. 즉, 각 출력값을 하나

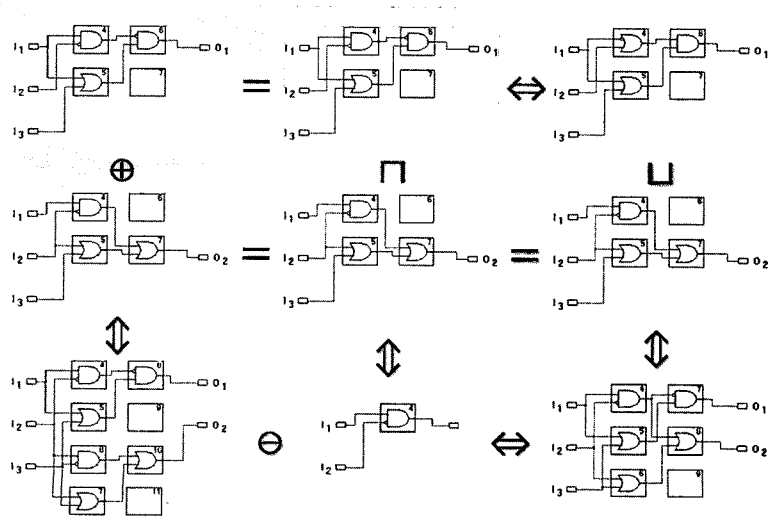


그림 4 표 2에서 정의한 연산자를 사용한 예. 여기서 '='는 동일한 회로, '⇔'는 입출력 패턴이 같은 동치 회로를 나타낸다.

의 출력으로 가지는 n_O개의 모듈로 나누어서 따로 진화시킨 다음 하드웨어의 특성을 고려하여 적절히 결합하여 목적 하드웨어를 얻는 방법이다. 이렇게 하면 크고 복잡한 하나의 하드웨어 대신 간단하고 작은 하드웨어 여러 개를 진화시키므로 진화에 필요한 시간을 단축시킬 수 있다. 그림 5에서 모듈로 나눠서 진화한 하드웨어는 앞서 소개한 ⊓연산과 ⊖, ⊕ 연산을 통해서 결합되어 목표 하드웨어를 구성한다.

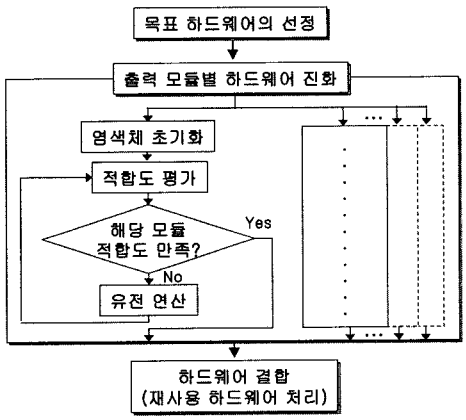


그림 5 모듈별 진화 하드웨어 설계 과정

제시하는 모듈별 진화가 얼마나 의미가 있는지 알아보기 위해 모듈화하기 전과 후의 염색체가 표현해야 하는 해영역의 크기를 살펴본다. 해영역의 크기는 하드웨어로 구성 가능한 모든 경우의 수를 의미한다. 따라서 염색체에 의해 표현 가능한 모든 경우의 수를 계산하면 알 수 있다. 모듈화 하지 않은 경우 해영역의 크기(L)는 다음과 같다.

$$L = L_1^{n_r} \times \prod_{n_c-1} L_2^{n_r} \times L_3 \tag{7}$$

여기서 n_r과 n_c는 논리기능셀의 행과 열의 수를 나타내고, L_1은 첫 번째 열에 위치한 논리기능셀이 표현 가능한 경우의 수를, L_2는 다른 열에 위치한 논리기능셀의 경우의 수를, L_3는 출력셀로 선택가능한 경우의 수를 뜻한다. 각 열에는 n_r개의 셀이 있으므로 L_1과 L_2는 n_r번 곱해지며, L_2는 첫 번째 열을 제외한 n_c-1열만큼 다시 곱해진다. L_1, L_2, L_3는 다음과 같다.

$$\begin{aligned} L_1 &= n_I \times n_I \times n_f \\ L_2 &= n_r \times n_r \times n_f \\ L_3 &= {}_{n_r}C_{n_O} \end{aligned} \tag{8}$$

여기서 n_I는 입력의 수, n_O는 출력의 수, n_f는 논리기능셀에서 선택 가능한 논리기능의 수를 나타낸다. 첫 번째 열은 두 개의 셀입력이 외부입력의 수 중에서 선택되기 때문에 n_I가 곱해지고, L_2는 다른 열의 셀이므로 n_r이 곱해진다. L_3은 마지막 열 n_r개의 셀 중에서 중복되지 않게 n_O개의 출력셀을 선택하므로 조합(Combination)이 사용되었다. L_1, L_2, L_3을 대입하고 식을 정리하면 해영역크기 L은 다음과 같다.

$$L = n_I^{2n_r} n_f^{n_c n_r} n_r^{2n_r(n_c-1)} \cdot \frac{n_r!}{(n_r - n_O)!} \tag{9}$$

위 식을 살펴보면 지수 부분을 차지하고 있는 변수가 모두 n_r과 n_c임을 알 수 있다. 즉, 논리기능셀 혹은 게이트의 수에 지수적으로 비례하여 해영역의 크기가 길어진다. 이 수식의 의미를 좀더 자세히 알아보기 위해 직접 값을 넣어 살펴보기로 한다. n_f를 본 논문에서 사용된 값인 10으로 두고, 입력수(n_I)와 출력수(n_O)는 2로 준 다음, n_c=n_r이라고 가정하여 게이트의 수(=n_c×n_r)에 대한 해영역 크기변화 그래프를 그려보면 그림 6과 같다.

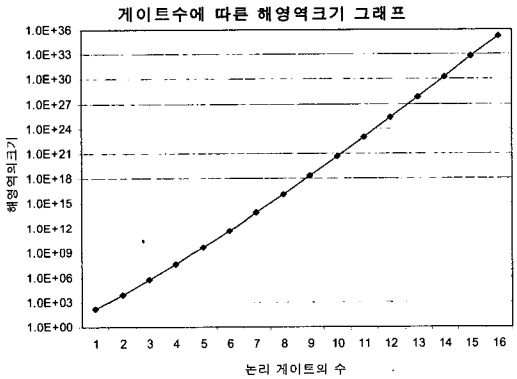


그림 6 게이트의 수에 따른 해영역 크기의 변화를 나타낸 로그 함수 그래프. 논리 게이트의 수는 n_r×n_c의 값을 의미한다. 그리고, n_r과 n_c의 값은 같다고 가정하였다.

그림 6을 보면 로그 함수 그래프임에도 불구하고 상당히 가파른 상승세를 보이고 있음을 알 수 있다. 즉, 논리 게이트의 수에 따라 해공간의 크기 혹은 염색체의 크기는 상당히 커진다는 것을 알 수 있다.

3.2 진화된 모듈회로의 결합

모듈별로 진화된 하드웨어는 목적 하드웨어로 구성하기 위해 결합해야 한다. 이때 단순 결합인 ⊕ 연산만 하는 것이 아니라, 수식 6에서와 같이 ⊓연산을 하여 중복된 하드웨어를 찾아낸 다음 ⊖ 연산을 하여 하나의 하드웨어만 재사용하도록 처리한다. 본 논문에서는 이 작업을 '재사용 처리'라고 부르기로 한다. 모듈 회로의 결합을 위해 수행된 과정은 다음과 같다. 원칙상으로는 수식 (6)에서와 같이 모듈을 하나씩 결합하면서 재귀적인 순서로 연산해야 하지만 본 논문에서는 편의상 한 번에

결합하여 중복 하드웨어를 제거하였다.

모듈의 집합에 관하여
① 모든 모듈의 단순 결합(⊕연산) 수행
② 중복 사용된 회로의 ⊖ 연산 및 재사용 처리

첫 번째 과정은 수식 6에서 ⊕로 표현된 단순결합 부분만을 처리하는 과정이다. 이 과정은 단순히 하드웨어의 크기를 적절히 늘리고 배열만 하면 된다. 다음 식은 첫 번째 과정만을 수행하여 얻는 F'을 나타낸다.

$$F' = f_{n_0-1}+f_{n_0-2}+\cdots+f_2+f_1 \quad (10)$$

두 번째 과정은 중복 부분을 제거하는 과정, 즉 재사용 처리하는 과정이다. 이 때 재사용 처리되는 부분이 많을수록 하드웨어 자원을 절약할 수 있고, 가능한 한 많은 기능이 재사용 처리가 되었을 때 최적화된 하드웨어를 얻을 수 있다. 하지만, 재사용 처리가 최대로 되는 모듈의 조합을 찾는 것은 쉽지 않다. 하나의 기능이 다양한 디지털 회로로 표현 가능하기 때문이다. 따라서 이러한 모듈 조합을 찾기 위해서 다음과 같은 과정을 수행하였다.

① 각 모듈을 C번씩 진화하여 C개의 하드웨어 획득
② 가능한 모든 모듈 조합에 대해서 재사용 처리 수행
③ 가장 최적화된 하드웨어 구조를 선택

위의 과정은 모든 모듈 조합의 가능성을 찾아보는 대신 모듈을 C개씩만 준비하여 재사용 결합을 시도하고 최적화된 회로를 찾는 방법이다. 본 논문에서는 C를 10으로 두었으므로 이 과정에서 ${}_{10}C_{n_0}$번의 결합을 시도한다. 이는 소모적이고 완벽한 방법은 아니지만 본 논문에서는 비교적 만족할만한 성능을 얻었다. 추후 좀더 효율적인 결합 방법을 연구할 필요가 있다.

4. 실험 결과

제시하는 방법의 성능 평가를 위해 2장에서 소개한 진화 하드웨어에 적용하여 실험을 하였다. 목표 하드웨어는 1비트 덧셈기와 2비트 덧셈기, 2비트 곱셈기로 하였으며, 각각 10번씩 진화하여 그 평균 및 추세를 비교하였다.

4.1 실험 환경

실험을 위해 설정되어 사용된 유전 연산 환경 변수는 표 3과 같다. 유전 연산자로는 룰렛휠 선택(roulette wheel selection) 방법과 일정교차(uniform crossover) 방법, 그리고 본 실험의 염색체에 맞게 변형한 비트플립 돌연변이(bit-flip mutation)를 사용하였으며, 엘리트 보존(elite preserving) 전략을 사용하여 좋은 해는 계속 유지되도록 하였다[11]. 그리고 원하는 해를 발견할 때까지 세대가 진행하도록 하였으며, 250만 세대까지만 진화가 가능하도록 제한하였다.

표 3 유전 연산 환경 변수 설정. 한계 세대는 진화 가능한 최대 세대를 의미한다.

유전 연산 환경 변수	값
개체 집단의 크기	100
선택률	1.0
교차율	1.0
돌연변이율	0.01
엘리트 유지 전략 사용	TRUE
한계 세대	2,500,000

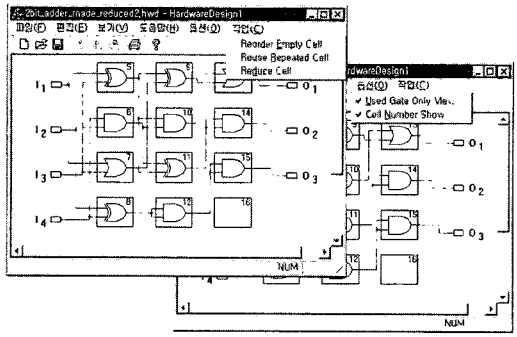


그림 7 진화되어 나온 디지털 회로를 알기 쉽게 표현하고 분석할 수 있는 어플리케이션

하드웨어의 진화를 잘 관찰하고 이해하기 위해서는 진화되고 있는 하드웨어의 구조를 알아보기 쉬운 디지털 회로 형태로 표현할 필요가 있다. 이를 위해 MFC (Microsoft Foundation Classes)를 이용하여 하드웨어 구조를 쉽게 관찰하고 조작할 수 있도록 지원하는 GUI (Graphic User Interface) 기반의 어플리케이션을 개발하여 사용하였다(그림 7).

4.2 실험 결과

모듈화하지 않은 직접진화 방법과 모듈별 진화 방법을 비교하기 위해서 각각 10번씩 실험한 평균 결과는 표 4와 같다.

이 표에 기록된 결과값은, 먼저 진화가 잘 되도록 넉넉한 게이트 크기를 주고 직접진화한 결과를 '직접진화(위)' 항목에 표기하였고, 이를 모듈로 나눠서 진화된 결과를 '부분 모듈 진화' 항목에 기록하였으며, 이렇게 얻어진 모듈을 단순히 결합연산('+')한 결과를 '모듈별 진화 후 결합(재사용 안함)' 항목에 나타내었다. 그리고, 각 모듈을 C(=10)번 진화해서 가장 최적의 결합이 적용된 하드웨어를 찾은 결과를 '모듈별 진화 후 결합(재사용함)' 항목에 기록하였으며, 모듈 진화 후 알게 된 최적화된 게이트 크기에 대한 직접진화의 결과를 '직접진화(아래)' 항목에 표기하였다.

실험 결과는 예상대로 모듈별로 진화한 경우가 직접

표 4 진화 방법별 하드웨어 진화 비교. 각 방법으로 10회의 진화 결과를 평균. 괄호 안의 값은 각 모듈의 진화된 결과값이다. 진화 한계 세대로 250만 세대를 설정하였고, 한계세대 내에 진화되지 않은 경우 '>>'로 표시하였다. '>'는 일부만 진화되었을 경우이다.

하드웨어	진화방법	재사용 적용	게이트수	평균탐색 세대수
1비트 덧셈기	직접 진화	No	2×2	36.7
		No	2×1	6.7
	모듈별 진화 후 결합	No	2×1	0
	(부분 모듈1)	(No)	(1×1)	(0)
	(부분 모듈2)	(No)	(1×1)	(0)
2비트 덧셈기	직접 진화	No	7×4	>417,454
		No	4×3	>>2,500,000
	모듈별 진화 후 결합	Yes	4×3	61,081
	모듈별 진화 후 결합	No	6×3	6,108.1
	(부분 모듈1)	(No)	(3×3)	(5,982.6)
	(부분 모듈2)	(No)	(2×2)	(125.1)
	(부분 모듈3)	(No)	(1×1)	(0.4)
2비트 곱셈기	직접 진화	No	7×3	>>2,500,000
		No	6×2	>>2,500,000
	모듈별 진화 후 결합	Yes	6×2	3711
	모듈별 진화 후 결합	No	7×2	371.1
	(부분 모듈1)	(No)	(2×2)	(23.8)
	(부분 모듈2)	(No)	(2×2)	(180.4)
	(부분 모듈3)	(No)	(2×2)	(166.6)
	(부분 모듈4)	(No)	(1×1)	(0.3)

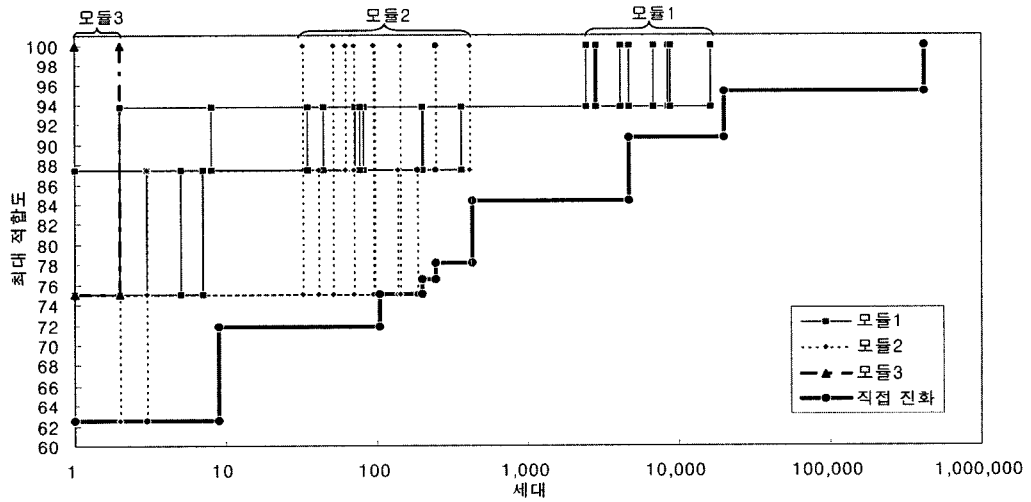


그림 8 2비트 덧셈기를 직접 진화와 모듈별로 진화하여 나온 세대별 최대 적합도 변화를 로그 함수로 나타낸 그래프. 모듈별 진화는 10번의 진화 결과를 모두 표시하였고, 직접 진화는 수렴을 성공한 경우만을 표시하였다.

진화한 경우에 비해 훨씬 적은 세대수를 나타내고 있다. 특히 모듈의 크기가 잘 분배되어 탐색 세대수가 크게 차이를 보이지 않는 2비트 곱셈기의 경우에는 모듈별 진화의 방법이 월등히 우수한 결과를 보여주고 있다. 직접 진화 방법으로는 250만 세대 이내에 진화조차 되지 않는 회로를 400세대 정도에 얻을 수 있었고, 재사용 처리를 위해 10회의 진화를 하더라도 4,000세대 정도만을 요구할 뿐이었다. 2비트 덧셈기의 경우에는 모듈화를 했음에도 의외로 많은 세대가 소모되었는데, 이는 나누어진 모듈의 크기가 고르지 않기 때문이다. 즉, 최적화 했을 경우 2비트 덧셈기는 4×3의 크기를 갖는데, 이것의 첫 번째 출력부분 모듈이 3×3 크기의 게이트를 차지함으로 인해 크기감소 효과가 적었기 때문이다. 그러나, 직접 진화가 7×3일 때 한번 진화를 성공하고, 4×3일 때 250만 세대 이내에 진화되지 않았던 것에 비해, 훨씬 진화가 잘 되고 있음을 알 수 있다. 1비트 덧셈기 실험에

서는 약 37배 정도 빠른 세대에, 그리고 2비트 덧셈기와 2비트 곱셈기에서는 400배, 6,000배 이상의 빠른 세대에 수렴되어 직접 진화에 비해 훨씬 효율적인 진화 성능을 보이고 있음을 알 수 있다.

직접 진화 방법과 모듈별로 진화하는 방법의 세대별 적합도 변화를 비교하기 위해 2비트 덧셈기의 진화 결과를 그래프로 표현하였다. 그림 8은 각각 10번씩 진화시킨 결과를 모두 표현한 그래프이다. 이때, 직접 진화 방법을 사용한 경우에는 250만 세대 이내에 수렴한 경우가 한 번밖에 없어서 그 경우만 표시하였다. 따라서 직접 진화 방법은 그래프에 표시된 것보다 훨씬 더 많은 세대를 요구함에 유의할 필요가 있다. 진화하는 양상을 보면 각 모듈이 직접진화한 하드웨어에 비해 훨씬 빠른 수렴을 하고 있음을 알 수 있다.

4.3 재사용 처리방법에 대한 고찰

실험에서 모듈별로 얻은 하드웨어를 결합하여 2비트 덧셈기를 얻는 과정을 좀더 자세히 살펴보자. 그림 9는 직접 진화 방법으로 전체를 한번에 진화시킨 2비트 덧셈기를 회로도로 나타낸 것이다. 그리고, 그림 10은 출력 부위별로 모듈을 나눠서 얻은 부분 하드웨어들을 나타낸 것이다. 그림에서 I_1과 I_2는 2비트 덧셈기 입력값 1을, I_3과 I_4는 입력값 2를 나타내며, O_1, O_2, O_3는 3비트의 전덧셈기 출력값을 나타낸다.

그림 10의 모듈들을 재사용 처리 없이 단순 결합하면 그림 11과 같은 형태의 회로가 완성된다. 그림 11에서 점선으로 표시된 부분이 부분회로들이다. 열과 행을 적절히 늘려서 결합하였으며, 출력 부분까지의 전달은 AND 게이트에 같은 입력을 통과하도록 하여 구현하였다.

모듈별로 진화해서 얻은 회로는 6×3의 게이트를 사용하고 있어서, 직접 진화해서 얻은 7×4 게이트 덧셈기 회로보다 더 효율적임을 알 수 있다. 이대로 사용해도 되지만, 중복된 하드웨어를 재사용 처리하면 더욱 최적화된 회로를 얻을 수 있다. 그림 11을 살펴보면 게이트 5번과 게이트 9번이, 그리고 게이트 6번과 게이트 8번이 같은 입력과 같은 논리기능을 가지고 있다. 이와 같은

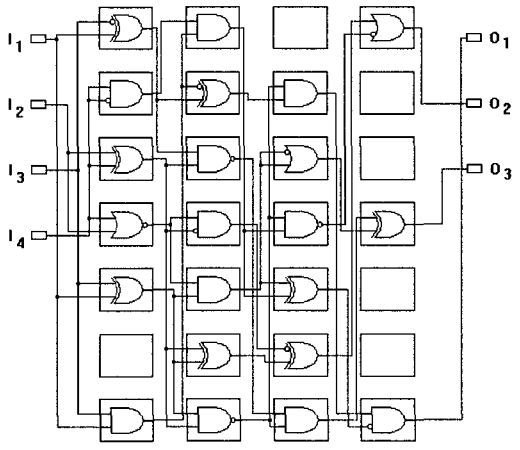


그림 9 직접 진화를 통해 얻은 7×4 게이트 배열의 2비트 덧셈기

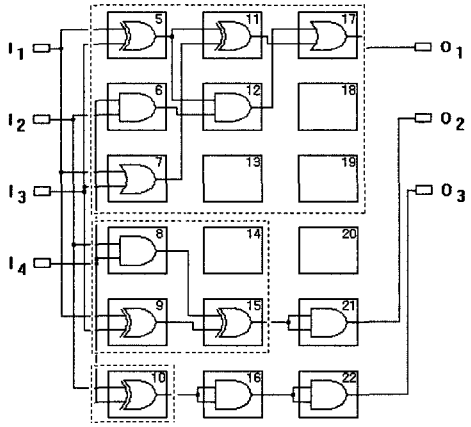
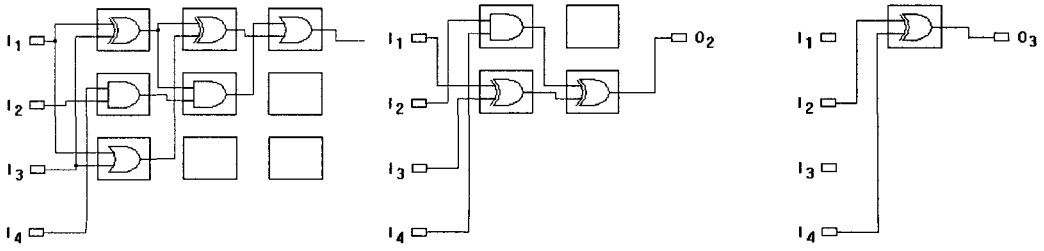


그림 11 모듈별로 진화시킨 뒤 단순 결합하여 얻은 6×3 게이트 배열의 2비트 덧셈기. 점선으로 표시된 부분은 결합에 사용된 모듈이다.



(a) 3×3 게이트 배열로 진화시켜 얻은 2비트 덧셈기의 O_1

(b) 2×2 게이트 배열로 진화시켜 얻은 2비트 덧셈기의 O_2

(c) 1×1 게이트 배열로 진화시켜 얻은 2비트 덧셈기의 O_3

그림 10 모듈별로 진화시켜서 얻은 2비트 덧셈기

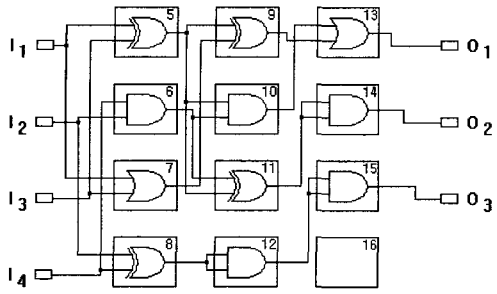


그림 12 모듈별로 진화시킨 뒤 결합하고, 재사용 처리하여 얻은 4×3 게이트 2비트 덧셈기

중복 사용된 게이트를 재사용 하도록 처리하여 간단한 회로를 완성한다. 그림 12는 재사용 처리된 회로이다.

재사용 처리 결과 상당히 간단한 회로가 완성되었다. 단순한 통과 기능으로 사용된 중복입력 게이트를 빼면 사용된 게이트의 수가 8개인 간단한 회로가 되었다. 모듈별 진화를 통해 직접진화로는 얻기 어려웠던 최적화된 하드웨어를 발견한 것이다.

이와 같이 재사용 처리를 하면 짧은 세대에 좋은 성능을 지닌 하드웨어를 얻을 수 있지만, 이와 같은 효과를 얻기 위해서는 모듈 사이에 재사용 가능한 회로가 많이 존재해야 한다. 그러나 한 번의 진화로 얻어진 모듈 사이에서는 그런 부분을 발견하기가 쉽지 않다. 따라서 모듈을 여러 개 찾아서 비교해야 할 필요가 있다. 이를 위해 실험에서는 C(=10)번씩 진화해서 얻은 모듈들을 조합해서 비교해보았다. 하지만, 10번 진화한다고 해서 항상 10개의 다른 하드웨어가 발견되는 것은 아니기 때문에 좀 더 효율적으로 수행하기 위한 방법이 필요하다.

진화 알고리즘을 통해 여러 해를 동시에 얻기 위한 방법으로는 병렬 GA를 이용한 방법과 종분화된(speciated) GA를 이용한 방법이 있다[12]. 병렬 GA는 앞서 10번의 시도를 한 것처럼 여러 GA를 동시에 수행하는 방법이고, 종분화 방법은 GA에 다양성을 높이는 기법을 적용함으로써 다양한 특성을 지닌 여러 종의 해를 한 번에 얻을 수 있게 해 주는 방법이다. 따라서 종분화 알고리즘을 이용하여 한번에 다양한 모듈을 한꺼번에 얻을 수 있다면 더욱 효과적인 모듈 진화가 될 것이다. 또한 종분화 알고리즘은 다양성이 높기 때문에 진화 하드웨어와 같이 해가 여러 개 분포하는 경우 세대를 단축시키는 효과도 있기 때문에 추가적인 이점도 기대할 수 있다[13]. 향후에는 이런 기법들을 적용하여 좀더 효율적인 모듈 결합 방법을 연구할 계획이다.

5. 결 론

진화 하드웨어는 환경 적응성이 뛰어나기 때문에 상당히 기대가 큰 분야임에도 불구하고 여러 실용상의 문제들로 인해 아직도 실생활에 제대로 활용되지 못하고 있다. 그러한 문제 가운데 하나가 바로 진화에 걸리는 시간이다. 본 논문에서도 모듈별 진화를 통해 얻었던 효율적인 크기의 2비트 덧셈기를, 직접 진화 방법으로 얻으려 여러 번 시도하여 보았지만 250만 세대 내에는 진화된 결과를 얻을 수가 없었다. 비슷한 연구를 한 다른 논문의 결과를 보면, 논리기능셀에 3입력 멀티플렉서 기능까지 사용하여 3비트 곱셈기를 진화시키는 데에도 거의 100만 세대 정도가 소요되고 있다[6].

본 논문에서는 이러한 문제를 해결하기 위한 방안의 하나로 하드웨어를 출력 부위 별로 모듈화하여 진화시킨 후 결합하는 방법을 제시하였고, 상당히 효율적인 진화가 가능함을 보였다. 정성적 분석에서 세대 단축을 예측하였고, 실제 실험에서도 상당 세대가 단축된 결과가 나타났다. 그리고 발견된 하드웨어도 직접 진화 방법에 의한 하드웨어보다 훨씬 간단한 최적화된 것이었다. 따라서 하드웨어에 필요한 진화의 시간을 줄이는 방법으로 상당히 유용함을 알 수 있었다.

하지만, 실험에서 몇몇 문제는 남겨둘 수밖에 없었다. 첫째로, 출력별로 모듈을 나누는 것은 출력이 여러 개 존재하는 경우에만 가능하다. 따라서 출력의 수가 적은 복잡한 하드웨어의 경우에는 이 방법을 사용해도 효과가 적다. 둘째, 진화될 하드웨어의 크기를 자동으로 설정할 수 없었다. 실험에서 셀의 크기를 설정하는 방법은 경험에 의존한 방법이었다. 모듈별로 나눈 경우에는 그나마 요구되는 셀의 크기가 크지 않아서 조금만 시도해봐도 되었지만, 직접 진화의 경우에는 최적의 하드웨어 크기를 예측할 수 없었다. 따라서 염색체의 크기가 자연스럽게 변하는 VGA(Variable GA)[14]와 같이 자연스러운 크기변화를 수용할 수 있는 염색체 구조의 연구가 필요하다. 마지막으로, 모듈별 진화에서 재사용을 통한 최적화 효과를 극대화시키기 위해서는 다양한 모듈을 한 번에 얻을 수 있는 방법과 재사용 방법의 다양화에 대한 연구를 계속해야 할 것이다.

참 고 문 헌

[1] M. Sipper et al., "A phylogenetic, ontogenetic, and epigenetic view of bio-inspired hardware systems," *IEEE Trans. on Evolutionary Computation*, vol. 1, no. 1, pp. 83-97, 1997. 4.

[2] X. Yao and T. Higuchi, "Promises and challenges of evolvable hardware," *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, vol. 29, pp. 87-97, 1999. 2.

[3] A. Thompson, *Hardware Evolution: Automatic Design of Electronic Circuits in Reconfigurable*

Hardware by Artificial Evolution, Springer-Verlag, 1998.

[4] W. Liu et al., "ATM cell scheduling by function level evolvable hardware," *Proc. of the First Int. Conf. Evolvable Systems: From Biology to Hardware*, pp. 180-192, Springer, 1996. 10.

[5] J. Torresen, "Increased complexity evolution applied to evolvable hardware," in *Smart Engineering System Design: Neural Networks, Fuzzy Logic, Evolutionary Programming, Data Mining, and Complex Systems, Proc. of ANNIE'99*, ASME Press, 1999. 11.

[6] V. K. Vassilev, "Scalability problems of digital circuit evolution: Evolvability and efficient designs," *Proc. of the Second NASA/DoD Workshop on Evolvable Hardware*, pp. 55-64, IEEE Computer Society, 2000. 7.

[7] John R. Koza, Forrest H. Bennett III, David Andre and Martin A. Keane, *Genetic Programming III: Darwinian Invention and Problem Solving*, San Francisco, CA: Morgan Kaufmann, 1999.

[8] John R. Koza, Martin A. Keane, and Matthew J. Streeter, The Importance of Reuse and Development in Evolvable Hardware, *NASA/DoD Conference on Evolvable Hardware*, pp. 33, July 09-11, 2003.

[9] J. F. Miller, et al., "Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study," *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science*, pp. 105-131, 1997.

[10] T. Higuchi 등, "유전자 학습에 의한 하드웨어 진화의 기초실험", 유전자 알고리즘, pp.365-393, 대청정보시스템(주), 1996.

[11] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison Wesley, 1989.

[12] K. Deb and W. M. Spears, "Speciation methods," *Evolutionary Computation 2:Advanced Algorithms and Operators*, ch. 14, Institute of Physics Publishing, 2000.

[13] 황금성, 조성배, "종분화를 이용한 다품종 하드웨어의 진화", 정보과학회 봄 학술발표 논문집(B), 제28권, 1호, pp. 307-309, 2001.

[14] I. Kajitani, et al., "Variable length chromosome GA for evolvable hardware," *Proc. of the 1996 IEEE Int. Conf. on Evolutionary Computation (ICEC'96)*, pp. 443-447, IEEE Press, 1996. 5.

[15] V. K. Vassilev et al., "Digital circuit evolution and fitness landscapes," *Proc. of the Congress on Evolutionary Computation*, vol. 2, pp. 1299-1306, IEEE Press, 1999.

[16] T. Higuchi et al., "Evolving hardware with genetic learning: A first step towards building a Darwin Machine," *Proc. of the 2nd Int. Conf. on the Simulation of Adaptive Behaviour (SAB92)*, pp. 417-424, MIT Press, 1993.

[17] T. Higuchi, et al., "Evolvable hardware," *Massively Parallel Artifical Intelligence*, pp. 398-421, MIT Press, 1994.

[18] M. Iwata, et al., "A pattern recognition system using evolvable hardware," *Proc. Parallel Problem Solving from Nature (PPSN IV)*, vol. 1141 of Lecture Notes in Computer Science, pp. 761-770, Springer Verlag, 1996.

[19] M. Murakawa, et al., "Hardware evolution at functional level," *Parallel Problem Solving from Nature IV*, vol. 1141 of Lecture Notes in Computer Science, pp. 62-71, Springer, 1996.

황 금 성

2000년 3월 연세대학교 컴퓨터과학과 졸업(학사). 2002년 3월 연세대학교 컴퓨터과학과 석사과정 졸업. 2004년 3월 연세대학교 컴퓨터과학과 박사과정 입학. 관심분야는 진화 하드웨어, 베이지안 기법, 개미집단 시스템

조 성 배

정보과학회논문지 : 소프트웨어 및 응용
제 31 권 제 1 호 참조